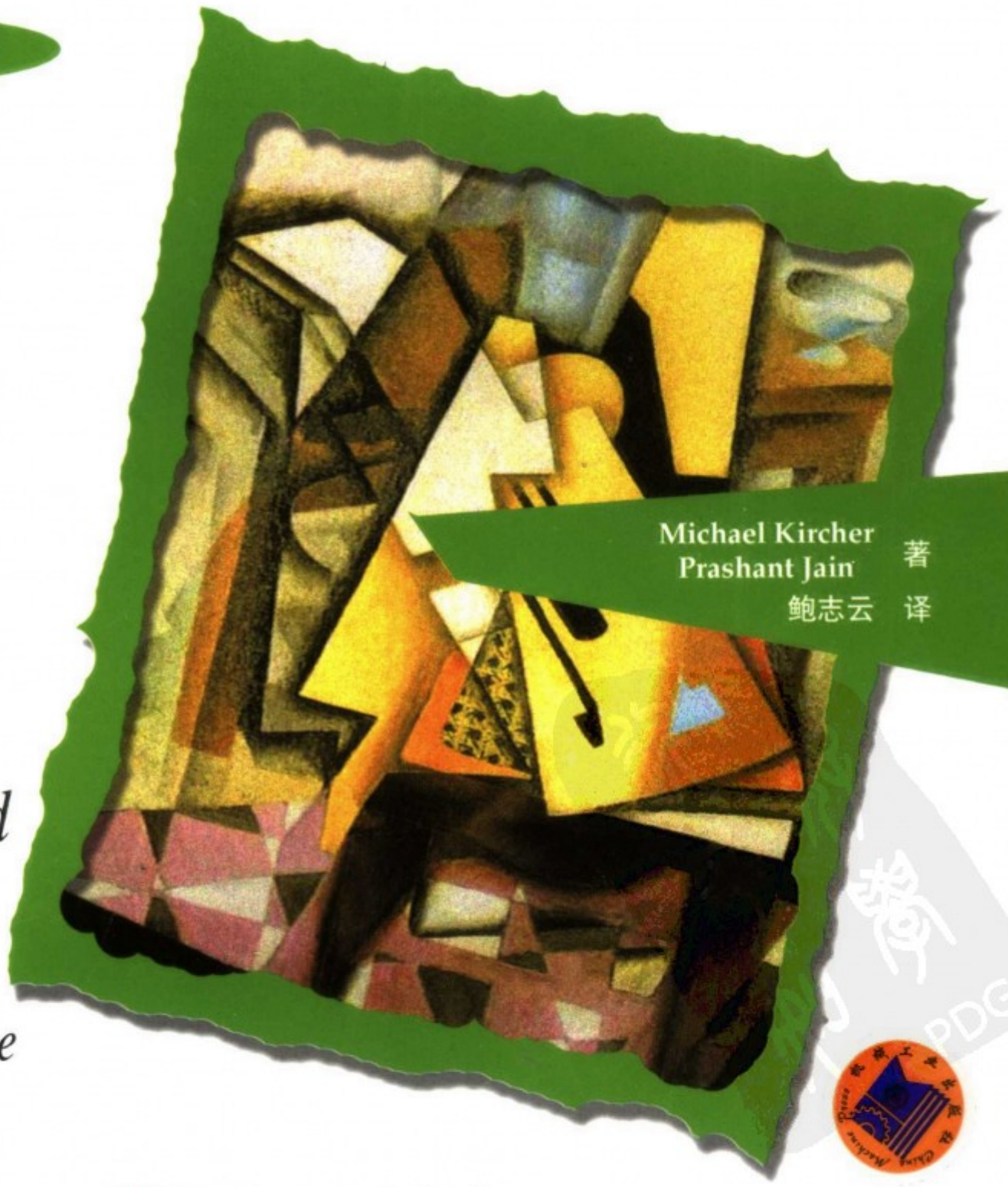
软 件 工 程 技 术 丛 书

HZ BOOKS

面向模式的软件体系结构

卷3

设计系列



Michael Kircher
Prashant Jain 著
鲍志云 译

Pattern-Oriented Software Architecture:
Patterns for Resource Management ,
Volume 3

机 械 工 业 出 版 社
China Machine Press

面向模式的软件体系结构卷3

在任何类型的软件中，有效管理资源都是至关重要的。从移动设备中的嵌入式软件，到大型企业服务器上的软件，有效地管理内存、线程、文件、网络连接之类的资源对于让系统可以正常且高效地工作都很重要。

我们经常在软件开发生命周期的后期才发现资源管理需求，而在这么晚的时候改变系统设计很困难。所以，在生命周期的早期执行这样的任务就很重要。因为属于不同领域的系统有不同的约束和需求，所以对某个特定系统或者配置很有效的方法对另一个系统就未必那么有效。

本书用模式来描述在系统中有效实现资源管理的方法。这些模式描述得很详细，使用了几个例子，并且和POSA前两卷一样，给出了如何实现它们的指导。此外，这一卷还对资源管理做了透彻的介绍，并给出了两个案例研究，分别把这些模式应用于自组网络计算和移动射频网络。这些模式归于不同的资源管理领域，涉及了完整的资源生命周期：获取、管理和释放。

作者简介

Michael和**Prashant**同POSA前两卷的作者紧密合作并且积极参与模式社区有好几年了。他们在各自的公司（西门子和IBM）中都在参与新技术和软件构架的研究与咨询。

上架指导：计算机/软件工程

ISBN 7-111-16983-2

9 787111 169833

封面设计：锡 林

HZ BOOKS

华章图书

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259, 68995264

读者服务信箱：hzedu@hzbook.com

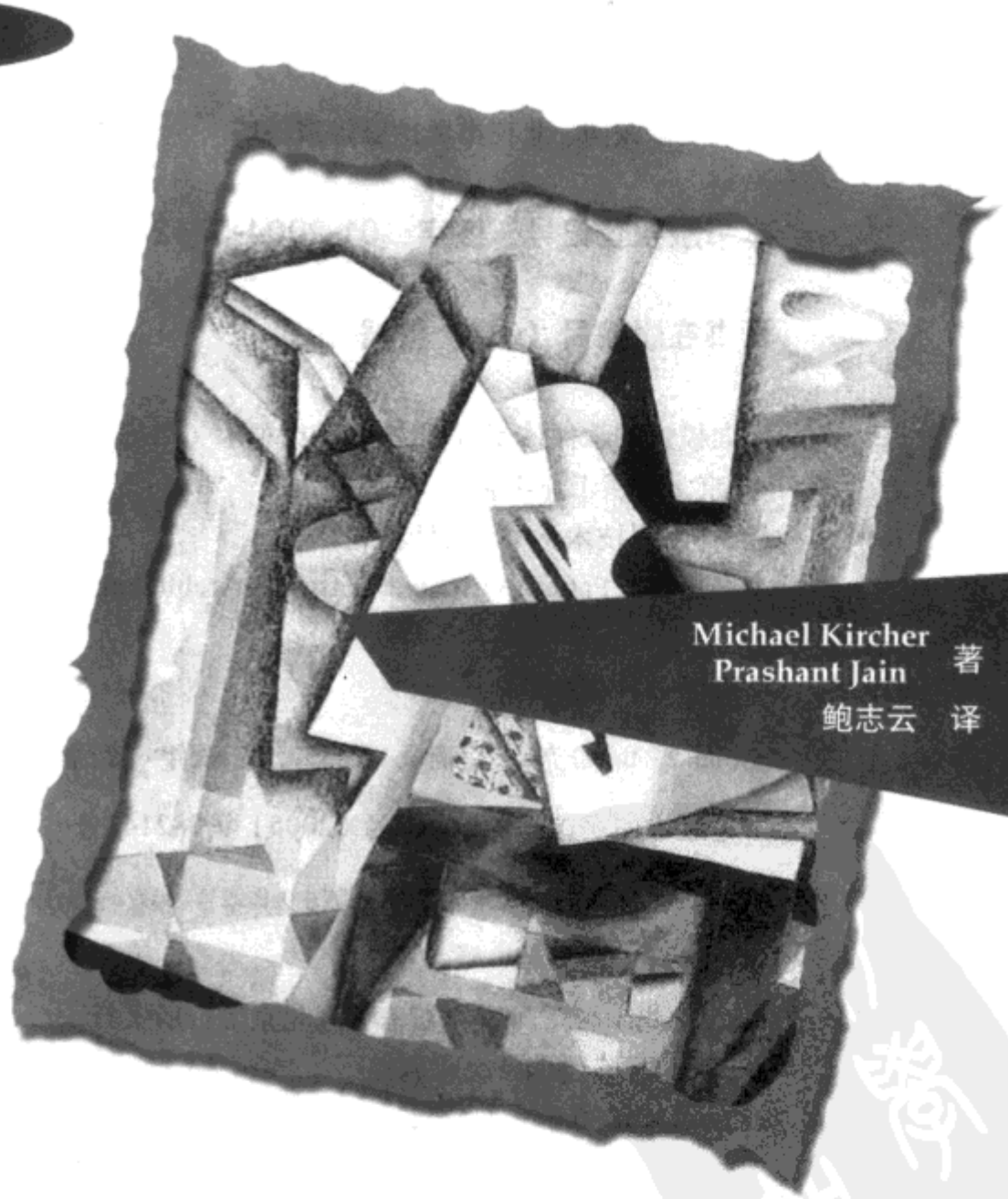
ISBN 7-111-16983-2/TP · 4379

定价：29.00 元

面向模式的软件体系结构

卷3

设计系列



Michael Kircher
Prashant Jain 著
鲍志云 译

Pattern-Oriented Software Architecture: Patterns for Resource Management, Volume 3

机械工业出版社
China Machine Press

本书用模式来展现在系统中实现有效的资源管理所需的技术，提供了对资源管理主题的介绍，讲解了资源获取、资源生命周期、资源释放中涉及的相关模式，这同典型的资源生命周期相对应。之后又列举了这些模式的两个案例分析，展示了这些模式如何用于现实世界中的应用程序，还展示了在这样的应用程序中这些模式如何相互关联。

本书适合软件构架师、设计者和开发者阅读，也可供计算机专业的学生参考。

Michael Kircher, Prashant Jain: *Pattern-Oriented Software Architecture: Patterns for Resource Management, Volume* 3 (ISBN: 0-470-84525-2).

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Copyright © 2004 by John Wiley & Sons Ltd.

All rights reserved.

本书中文简体字版由约翰-威利父子公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2004-5742

图书在版编目（CIP）数据

面向模式的软件体系结构　卷3/克车尔（Kircher, M.），斋尔（Jain, P.）著；鲍志云译. －北京：机械工业出版社，2005.9

（软件工程技术丛书　设计系列）

书名原文：Pattern-Oriented Software Architecture: Patterns for Resource Management, Volume 3

ISBN 7-111-16983-2

Ⅰ.面...　Ⅱ.①克...　②斋...　③鲍...　Ⅲ.软件－系统结构　Ⅳ.TP311.5

中国版本图书馆CIP数据核字（2005）第082142号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：李云静

北京京北制版厂印刷·新华书店北京发行所发行

2005年9月第1版第1次印刷

787mm × 1020mm 1/16 · 11.75印张

印数：0 001-4 000册

定价：29.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

译者序

这本书是专讲资源管理模式的。这在模式类书籍中是较为少见的。而资源管理这个话题却又是如此地重要，特别是对于24×7运行的大型系统，资源管理的成败对系统的成败有极大的影响。

很多读者可能知道，现在敏捷方法很流行，人们开始觉得很多设计都可以在后期通过重构的方式加入，而不需要一开始就严密地思考并决定。关于这些设计的模式，实在可以“妙手偶得之”，而不必一开始就有意识地去用。

但是，也有一些涉及系统全局的方面是很难通过重构的方式逐渐引入或者改变的（或者后期引入带来的变动太大，使得这样做显得不划算）。这些方面包括对国际化和本地化的支持、对多线程的支持、安全性、可移植性等等。这其中也包括了资源管理策略。

所以，在设计系统之前，有必要先对资源管理模式有个全盘的了解，这样在设计时才能有意识地运用最合适的资源管理方法。

在本书中，我个人最喜欢的是第7章的案例分析，该章描述了设计一个UTMS移动网络系统（包含基站、RNC、OMC）时遇到的挑战，以及如何从“模式弹药库”中找到合适的弹药来解决这些挑战。读来感觉淋漓尽致。这一章篇幅不长，但信息量很大，因为所有的细节都隐藏在模式名的背后。所以，为了读懂第7章，必须先读第2、3、4章。（事实上，这3章也可以当做资源模式管理词典，供你在工作中备查。）

其实，这本书还有模式地图的作用，因为书中不仅提到了资源管理模式，还以此为主干展开，广泛引用了模式社区的大量其他著作和模式。这也算是一个额外的好处吧。将单个模式融入更宏大的模式地图（或者说在“模式语言”的语境中）来讲述，这也是POSA系列的特色之一。

相信此译本可以带给读者愉快的阅读体验。若阅读过程中发现任何问题，可以发信到：wesley.bao@acm.org。

鲍志云

2005年7月

Frank Buschmann序

我从没想到会发生这样的事情：POSA系列新的一卷的作者竟没有一个属于“Party of Five”[⊖]。但我为此感到骄傲！因为本书意味着新一代的优秀软件工程师成长起来了，并且在积极地把他们的经验贡献给模式社区；还因为本书意味着模式的理念依然在繁衍、成长。当然，我也为我们1996年的POSA愿景依然在激励模式创作者们增强、完善、发展我们的材料而感到骄傲。

POSA这一卷所选的主题——资源管理，几乎对所有软件系统的成功都至关重要。或许这看起来显而易见，但仅仅在十年前，很多开发者还认为只有在嵌入式系统领域我们才需要关心资源管理问题。在桌面和企业领域，只有很少的开发者注意这个主题。为什么要关心像内存这样的资源呢？如果不够用了，只要升级机器就行了。我必须承认，在我职业生涯的早期阶段，我也持类似的观点，以为资源是取之不尽、用之不竭的。我那时的观点是多么错误啊！幸运的是，我曾被自己的错误绊了一跤，并很快吸取了教训。

今天，很少有开发者会忽略资源管理的重要性。随着基于组件的系统、应用服务器的出现，随着运行于我们计算机上的应用程序的复杂性和规模的增长，人们开始意识到，小心地管理资源会对软件系统的质量产生多么大的影响。内存、CPU运算能力、线程、连接等资源都是有限的，即便对于最强大的系统也是如此。但人们期望，即便会有大量用户同时访问这些服务器，它们也应能为所有用户提供高质量的服务。这一冲突只能通过明确、谨慎地管理服务器资源来解决。“资源”这个术语并不仅限于内存或连接这样的底层概念。在今天的网络计算世界中，资源还可以包括被远程客户应用程序使用的组件和服务。多个客户应用程序常常会竞争访问这些组件和服务。确保所有的客户都获得足够好的服务也是恰当地管理资源所应关注的内容。

不过，认识到资源管理的重要性和把资源管理做好是两码事。有效管理资源很困难，很有挑战性。如果你能做好，那么你的应用程序将具有高效率、稳定性、可伸缩性、可预期性和可访问性。如果你没有做好，那么在最好的情况下你的应用程序将提供很有限的操作质量，在最糟的情况下应用程序会失败，就那么简单。让容器来做资源管理并不是万能的解决方案，因为很多软件系统不能承受这样的基础设施的开销。即便可以用容器，你也需要理解容器是如何管理资源的，这样才能创建高质量的软件。

⊖ 指前面两卷的5个作者。与《Design Patterns: Elements of Reusable Object-Oriented Software》的4个作者“Gang of Four”相映成趣。——译者注

很多应用程序使用了容器，但还是失败了，原因仅仅是缺乏理解。

那么如何去理解呢？去哪儿找关于资源管理的挑战、应对挑战的解决方案以及解决方案的实施建议的资料呢？你拿着的这本书，POSA系列的第3卷，就是这样的资料之一。它展示了多年积累下来的资源管理经验和方案。大量的著名应用程序和中间件都证明了这些经验和方案的质量。把这些经验和方案用模式的方式表示出来可以让每个软件开发者都能接触它们。新手可以学会资源管理需要关注的基本问题以及基本的解决方案，专家则能交叉检查并评估各种方案，并阅读特定方案的细节。我还不曾听说有跟此书同样详尽的关于资源管理的资料。

在一开始我说过，我为本书感到骄傲。你读过之后就会知道为什么。

Frank Buschmann
西门子公司技术部门

Steve Vinoski序

如果你曾做过较长时间的软件开发工作，你几乎肯定会经历过我称做“沿计算机内存小道回溯”的事情。这样的事情通常发生在开发者们相聚在社交场合的时候，比如在一起吃午饭时。开端总是无意的，一位开发者说起他最近解决的一个特别困难的问题。然后，另一个开发者不甘示弱，开始讲述自己征服另一个更难问题的故事。然后，其他人的每个故事中的情形都会更困难一些。直到最后，元老们开始讲他们那时用打孔卡片或者开关面板来编程的只有几个字节内存的古董机器的故事。我在等有没有哪一天会有人跟我说当他/她开始编程的时候，机器语言只有0没有1!

开发者能够用上面的方式来攀比，因为编程天生就需要做出很多取舍。应用程序需要和别的程序分享计算资源。内存空间和磁盘空间不是无限的。CPU在一秒钟只能处理有限数量的指令。磁盘和设备输入输出可能会花比较长的时间。建立数据库连接和网络连接可能需要耗费很多时间和资源。在电子计算的历史长河中，硬件、操作系统、中间件和应用程序都取得了长足的进步。但是，虽然进步世人皆睹，但编程依然是关于做出正确取舍以便尽可能提高整个计算解决方案的效率的艺术。

所有的应用程序都会管理某种资源，比如内存、网络或数据库连接、线程。但是，在资源管理方面编写并优化得好的程序和做得不好的程序之间有显著差别。对偶尔运行一小段时间的程序（比如基本的命令行工具或者图形配置界面）而言，忽略资源管理不算什么问题。但若要开发需要长期运行的健壮、高性能、可伸缩的应用程序（比如Web服务器、应用程序服务器、通告系统），那么对资源管理缺乏关注就意味着失败。

本书中的模式都是关于资源管理的。概括地讲，资源可以获取、管理、释放。本书中展示的模式就用于以上3个领域。这些是对长期运行的应用程序的性能、规模、可伸缩性、延续性具有深远影响的主要领域。例如，很多应用程序都从堆上分配内存。对像Web服务器或应用服务器这样的程序来说，在请求处理代码路径中每一个从堆上分配内存的操作都降低了服务器的性能和可伸缩性，因为调用堆管理器以及获取和释放用来保护对堆的同步访问的互斥锁的代价高昂。这样的应用程序可以使用Partial Acquisition这样的模式，在进入请求代码路径之前就尽可能多地执行获取资源操作；并且使用Caching或者Pooling模式，以便保留资源用于下次请求，而不是释放资源并且下次重新分配。即便是有经验的开发者也会对资源管理模式的正确组合带来的性能和可伸缩性提升感到惊喜。

本书继承了POSA强调实际方案的传统。我特别喜欢的是书中每个模式都包含了详细描述实现问题的小节，而且每个模式都包含一个详尽列出这个模式已知应用的小

节。此外，还有两章讲述了具体的案例研究，展示了这些模式如何用于现实世界中的应用程序，还展示了在这样的应用程序中这些模式如何相互关联。归根结底，模式是对在真实应用程序中已经证实有效的方法的描述。作者在书中有效地对模式追根溯源并分析其影响，从而确保这些模式保持了同现实世界间的重要联系。

软件模式通常会帮助我们决定在构架和设计中做出怎样的取舍，以及在何处取舍。毕竟编程工作是由人来做的，而模式可提升抽象层次，并有助于团队沟通他们的构架和设计。我从事中间件构架、设计和实现的工作已经有很长时间了，其间我多次成功地应用Michael和Prashant在本书中展现的模式。但遗憾的是，这是在这些方法被表示成模式之前，这意味着我和其他团队成员花了大量的时间来设计出我们自己的变体、实现它们、评估它们以决定取舍，并且基于评估结果优化和调整它们。现在，Michael和Prashant已经清晰而详尽地把这些方法用模式语言表达出来了，你和你的团队成员可以更简单地讨论它们，并理解它们的好处，判断在什么样的情形下使用它们最好，并理解实现它们时你需要注意的方方面面。

当你已经用关于这些资源管理模式的知识和好的性能评估工具“武装”起来时，你可能会惊喜地发现你可以那么快、那么容易地大幅度提高应用程序的性能和可伸缩性。拥有了这样的技能，在下一次“沿计算机内存小道回溯”的时候，你的故事会让元老们都吃惊的。

Steve Vinoski
IONA Technologies公司产品创新部门首席工程师

前　言

这本书是关于软件系统中的资源管理模式的。这些模式为软件构架师和开发者们在试图为软件系统提供有效且高效的资源管理方法时常常遇到的问题提供了解决方案。高效地管理资源，对于任意类型的软件的执行都是至关重要的。从移动设备中的嵌入式软件，到大型企业服务器中的软件，有效地管理内存、线程、文件或者网络连接之类的资源对于让系统正常并且有效工作都很重要。

Pattern-Oriented Software Architecture（POSA）系列的第1卷[POSA1]广泛地引入了软件设计和构架中的通用模式。第2卷[POSA2]则专注于建立复杂的并发和网络软件系统与应用程序的基本模式。现在本卷则用模式来展现在系统中实现有效的资源管理所需的技术。

书中的模式详尽展开，并且会有一些例子。同POSA的前几卷一样，本书为读者提供了如何实现这些模式的指导。此外，本书还对资源管理作了透彻的介绍，并且包含了两个案例分析，在案例中这些模式应用在两个不同的领域。书中展示的模式独立于任何实现技术（比如.NET、Java或者C++），虽然例子是用Java和C++给出的。模式根据资源管理的不同领域来分类，因此涉及了资源的完整生命周期：资源获取、资源生命周期和资源释放。

本书中的模式广泛地覆盖了资源管理领域。我们在几年前开始基于自己创建很多不同软件系统的经验记录这些模式。大多数模式都曾经在主要的会议上展示过或者讨论过。但是，我们觉得还缺少这样的努力，把这些模式以模式语言的形式放到一起，并以可以把模式语言应用到多个领域的方式将其展示出来。

资源管理的范围很广。随着新技术的出现，系统设计者和开发者面对的资源管理的挑战也时刻在变化。我们预计，随着时间的推移，人们会发现并记录更多的资源管理模式。本书的“结语”那一章谈论了发展资源管理模式语言还需要进行怎样的努力。

读者群

本书是为所有的软件构架师、设计者和开发者而写的。他们可以用本书中描述的模式来解决通常每个软件系统都会遇到的资源管理的挑战。

本书对计算机科学专业的学生也很有用，因为它可以提供对资源管理的现有最佳实践的概览。

本书结构

本书分成两部分。第一部分提供了对资源管理的主题以及资源管理的模式的介绍。第一部分展示的模式分成三章：资源获取、资源生命周期、资源释放，这同典型的资源生命周期相对应。本书的第二部分把这些模式应用于两个案例。

本书的第一部分从问题领域的角度来看待资源管理，而第二部分则从应用程序领域的角度来看待资源管理。本书中的模式不是孤立的。事实上，在我们覆盖资源管理模式的整个过程中，我们大量引用了其他相关的值得注意的模式。对于每个这样的模式，我们都在“引用到的模式”一章中包含了摘要介绍。[⊖]

本书包含了很多使用模式的例子。虽然模式用的都是单独的例子，但讲案例的每一章都是用一个特定领域的例子来把所有的模式联系起来。这样就起到了一个完整的例子的作用，把特定领域中的问题和单个模式的使用放到了一起，以解决所提出的问题。这种方法让我们可以证明模式的广泛可用性，同时还展示了它们如何联系到一起。

第1章“导引”正式引入了软件系统中资源管理的话题，并且定义了它的范围。这一章描述了为何有效地管理软件系统中的资源那么难。这一章还介绍了模式，并展示了如何用模式来应对资源管理的挑战。

第2章“资源获取”描述了解决对资源获取有影响的作用力的模式。资源在使用前必须先获取。但是，获取资源不应该降低系统性能，也不应该造成任何瓶颈。对于大型资源或者只有一部分可用的资源，资源获取策略的调整是至关重要的。

第3章“资源生命周期”描述了解决影响资源生命周期的作用力。资源可以有很不相同的生命周期。例如，有的资源会被频繁且密集地使用，还有些资源则只会被用到一次。当我们不再需要某一资源时，就可以释放它。但是，判断何时释放资源并不是那么简单。对资源生命周期的显式控制可能既令人厌烦又容易出错。为了解决这一问题，我们需要自动资源管理技术。此外，在某些特定的构架（比如分布式系统）中，可能多个资源必须共同工作以达成更大的共同目标。每个资源都有可能由自己的控制线程来管理，那么就需要协调多个资源的协作和整合。

第4章“资源释放”描述了处理资源的有效释放的模式。当资源不再被需要时，就需要返回给资源环境，以优化系统性能并让其他使用者可以获取资源。但是，如果被释放的资源需要被相同的资源使用者再次使用，那么就必须重新获取资源，这就会影响性能。这里的挑战是如何找到正确的平衡并决定何时释放资源。此外，对资源管理操作（比如释放资源）的显式调用很令人厌烦。如何尽量减少资源管理的开销，同时还确保高效率和高可伸缩性呢？

在第5章“资源管理准则”中，我们展示了进行资源管理的准则，这为在特定领域中高效地使用资源管理模式语言提供了指导。

第6章“案例分析——自组网络计算”展示了如何用我们描述的模式来创建自组网络应用程序并满足它的资源管理需求。

第7章“案例分析——移动网络”把所有的模式都整合进模式语言，并用这个模式语言来解

⊖ 习惯上我们使用模式的英文名，所以本书正文中凡模式名都不译出。但对本书介绍的资源管理模式，都在模式标题下给出了参考译文，并在“引用到的模式”中给出了书中出现的其他模式的参考译文。——译者注

决一个电信领域的案例的需求。

在第8章“模式的过去、现在和未来”中，Frank Buschmann审视过去在POSA前一卷中做出的对“模式将走向何方”的预言，并进一步分析模式现在走到了哪一步，然后预测模式的未来。

第9章“结语”为全书的内容收尾。该章包含了对资源管理领域将来可能会有什么样的成果的分析。

“引用到的模式”简洁地描述了我们引用到的模式，“符号表示法”则解释了我们在书中用到的所有符号表示法。

若读者需要补充材料，那么建议访问我们的Web站点http://www.posa3.org。这个站点包含了我们在模式中所展示的所有源代码，还包含了对模式自身的更新。随着时间的推移，资源管理模式语言所新增的内容也会包含在这个站点上。

如果你有任何评论、建设性的批评，或者改进建议，请用e-mail发到authors@posa3.org。

导读

本书的写法适合从头读到尾。但是，如果你知道要达到什么目的，那么就可以选择自己的阅读步骤。对于这种读法，下面的提示可能有助于你决定选择哪些主题以及用什么顺序来阅读：

- 若要知道单独的模式在实践中如何使用，以及它们如何协作，可以先读模式的“问题（Problem）”和“解决方案（Solution）”小节，然后读第6章和第7章的案例分析。
- 若想感受模式语言的广泛可用性，可以阅读第2章到第4章每个模式的摘要以及“已知应用（Known Uses）”小节。
- 若要知道资源管理模式语言如何同现有的模式整合（特别是那些涉及资源管理领域的模式），可参见第1.5节“相关工作”。

我们在扉页中概括了模式语言，列出了本节介绍的所有模式的摘要，从而提供了模式语言的概览。我们还展示了一幅描述模式之间关系的模式地图。

致谢

能向那么多支持我们写作本书的人表示谢意，我们感到很愉快。首先，我们要感谢我们的指导者Charles Weir，以及为本书审稿的Cor Baars、Frank Buschmann、 Fabio Kon、Karl Pröse、Christa Schwanninger、Michael Stal、Christoph Stückjuergen、Bill Willis和Egon Wuchner。还要特别感谢硅谷模式小组的Trace Bialik、Jeffrey Miller、Russ Rufer和Wayne Vucenic为本书作出了极大贡献。事实表明，通过Wiki在因特网上协作，获取来自硅谷模式小组的反馈是非常有效的。

我们还要感谢我们的EuroPLoP和PLoP审稿人，他们详尽地审阅了每个模式。他们是Pascal Costanza、Ed Fernandez、Alejandra Garrido、Bob Hanmer、Kevlin Henney、Irfan Pyarali、Terry Terunobu、John Vlissides和Uwe Zdun。我们还要感谢每一章的审稿人：Roland Grimminger、Kevlin Henney、Michael Klug、Douglas C. Schmidt、Peter Sommerlad和Markus Völter。Kevlin也为我们提供了支持，他提供了很多章节引用，审阅我们的图中对UML的使用，并给出了对空格使用的建议。

我们还要感谢Kirthika Parameswaran，我们跟他一起为实现JinACE[KiJa04]⊖的想法而工作。JinACE是一个类似Jini[Sun04c]的框架，但用C++实现。我们的工作激励我们对自组网络计算的概念进行更深的探索，并最终发现了它背后的模式。我们要感谢她通过长长的e-mail交互所进行的创造性讨论。

我们很感激Douglas C. Schmidt，是他鼓励我们从模式语言的角度来看待我们对JinACE的研究。随后浮现的模式激励我们开始对资源管理模式语言的研究。

此外，我们还要感激西门子公司技术部的模式团队：Martin Botzler、Frank Buschmann、Michael Stal、Karl Pröse、Christa Schwanninger、Dietmar Schütz和Egon Wuchner。

我们还要感谢John Wiley & Sons出版社的联系人。他们为本书提供了很棒的支持。他们是Gaynor Redvers-Mutton、Juliet Booker和Jonathan Shipley。我们的技术编辑Steve Rickaby对于我们完善此书提供了很大帮助，跟他合作很愉快。

最后，我们还要向Frank Buschmann致以特别的感谢。他不仅审校了本书，还贡献了“模式的过去、现在和未来”那一章。是他带来的灵感，是他的鼓励，帮助我们完成了此书。

⊖ 此种方括号的内容与本书后面的“参考文献”相呼应，下同。

作者简介

Michael Kircher

Michael Kircher是德国慕尼黑的西门子公司技术部门的高级软件工程师，他的主要关注领域包括分布式对象计算、软件构架、模式、极限编程和创新环境中的知识员工管理。他曾作为顾问和开发者参加过很多项目，涉及了西门子的多个不同业务领域，为分布式系统开发软件，其中包括UMTS基站的软件、自动邮政系统、用于工业及电信系统的运营维持软件。

在学生时代，他是Douglas C. Schmidt领导的研究组的成员，在这个研究组中他参与了TAO（The ACE ORB，一种实时CORBA的实现）的开发。就是在那时，他发现了模式的世界。在实现TAO的高效多线程分派机制时，他同Irfan Pyarali合作创作了他的第一个模式：Leader/Followers模式。

近年来，他在多个会议上发表了关于模式、分布式系统软件构架、极限编程方面的论文。他还同Prashant一起在OOPSLA和EuroPLoP这样的会议上组织了一些关于本书中一些内容的研讨会。

在闲暇时间，Michael喜欢和家人一起享受大自然的乐趣，或者步行或者骑车。他最喜欢的休闲活动是穿上自制皮衣并带上猎狗Ella去观察野生动物。

Prashant Jain

Prashant Jain是IBM设在印度德里的研究实验室的技术成员。他的主要关注领域包括分布式系统、电子商务、软件构架、模式及极限编程。在IBM，他的研究方向是电子商务领域的前沿技术。

Prashant从美国圣路易斯的华盛顿大学获得计算机科学硕士学位。他对设计模式的兴趣就是在那里形成的。1996年，他与导师Douglas C. Schmidt共同创作了他的第一个模式。从此以后，他就积极参加模式社区，创作并提交模式，并在OOPSLA和EuroPLoP这样的会议上组织模式研讨会。

他热衷于旅行，曾在印度、日本、美国、德国工作和生活过。他曾工作于西门子、富士通网络通信、柯达医学影像系统等公司。除此之外，他还积极参与华盛顿大学分布式对象计算中心的工作。

在闲暇时间，Prashant喜欢旅游和野餐，看电影，游泳。但他最珍爱的事情是和他4岁的女儿Aanya进行逻辑对话。Aanya常常能说得他哑口无言!

目　录

译者序
Frank Buschmann序
Steve Vinoski序
前言
作者简介
第1章　导引 ……1
1.1　资源管理概览 ……2
1.2　资源管理的范围 ……3
1.3　模式的使用 ……5
1.4　资源管理中的模式 ……5
1.5　相关工作 ……7
1.6　模式格式 ……9
第2章　资源获取 ……11
2.1　Lookup模式 ……11
2.2　Lazy Acquisition模式 ……22
2.3　Eager Acquisition模式 ……31
2.4　Partial Acquisition模式 ……40
第3章　资源生命周期 ……49
3.1　Caching模式 ……49
3.2　Pooling模式 ……57
3.3　Coordinator模式 ……67
3.4　Resource Lifecycle Manager模式 …78
第4章　资源释放 ……91
4.1　Leasing模式 ……91
4.2　Evictor模式 ……104
第5章　资源管理准则 ……113
第6章　案例分析——自组网络计算 …115
6.1　概览 ……115
6.2　动机 ……116
6.3　解决方案 ……117
第7章　案例分析——移动网络 ……123
7.1　概览 ……123
7.2　动机 ……126
7.3　解决方案 ……126
第8章　模式的过去、现在和未来 ……139
8.1　过去的四年 ……139
8.2　模式的现状如何 ……142
8.3　模式明天将走向何方 ……142
8.4　关于模式未来的简短声明 ……145
第9章　结语 ……147
引用到的模式 ……149
符号表示法 ……153
参考文献 ……159
致谢 ……170

第1章 导 引

"人们试图设计傻瓜型产品时常犯的一个错误是低估了傻瓜的智商。"

——Douglas Adams

重点

资源是供给有限的实体。存在请求者，即资源使用者，需要该实体来执行特定功能；还有机制，即资源提供者，用于提供被请求的实体。在软件系统的背景下，资源可以包括内存、同步原语、文件句柄、网络连接、安全令牌、数据库会话、本地及分布式服务,等等。资源的范围很广，从重量级对象（比如应用服务器组件[VSW02]）到细粒度的轻量级对象（比如文件句柄）都可以。

有时候，判断资源是什么有点难度。例如，在编程环境中，JPEG或者GIF文件之类的图像经常被称做资源。但实际上，图像并没有定义获取和释放语义，代表图像的数据才是资源。因此，更准确的表达是把图像使用的内存看做资源，装载图像时需要获取内存；当图像不再需要时应该释放内存。

给资源分类的方法有好多种。最简单的分法是把资源分成可重用和不可重用这两种。我们通常从资源提供者那里获取可重用资源，使用它，然后释放它。一旦资源被释放，它们就可重新被获取和使用。可重用资源的一个例子是内存，操作系统会负责分配和释放内存。其他的例子还有文件句柄和线程。可重用资源是最重要的资源形式，因为资源提供者通常只有数量有限的资源，重用未被消耗的资源是合理的。与此相反，不可重用的资源会被消耗掉，所以获取后要么不再释放，要么隐式释放。不可重用资源的一个例子是计算网格中的处理时间[Grid04]，一旦获取了处理时间并使用它，那么时间就被消耗掉了，不能返回给提供者。

给资源分类的另一种方法是基于资源如何被访问或使用。资源一旦被获取后，既可以被多个用户使用，也可以只给一个用户使用。被多个用户同时访问的资源例子有服务、队列、数据库。如果可以被多个用户同时访问的资源具有可改变的状态，那么对资源的访问就需要被同步。相反，如果资源没有状态，那么访问就不需要同步。J2EE EJB[Sun04b]应用程序服务器的无状态Session Bean就是不需要同步的资源的例子。而网络通信socket则是需要同步的资源的例子。可以被多个用户同时访问的资源不需要由每个用户来显式获取。资源使用者可以共享到资源的引用，这样，其中一个资源使用者在一开始获得资源引用，其他使用者就可以用现成的了。

与此相反，如果资源只能被一个用户使用，那么它就称为独占式资源。服务的处理时间就是独占式资源的例子。处理时间可以被看做是服务所提供的不可重用独占式资源。服务成了资源提供者，资源使用者可以从它那里获取处理时间。服务本身可看做另一种资源。获取服务意

味着获取处理时间。

独占式资源可以是可重用的，也可以不可重用。但若资源不可重用，那么它一定是独占式的，因为它只能被一个资源使用者使用。此外，如果一项独占式资源是可重用的，那么它一定是串行可重用的，即资源随着时间推移通过资源提供者得到重用。

给资源分类常常既不容易又不是很有意义。例如，同服务的处理时间相对照，CPU时间也可被看做一种有价值的资源。从某种方面上说，我们可以把CPU资源看做不可重用的独占性资源，由线程取得，线程是资源使用者。但在现实中，CPU时间不受任何应用程序的控制，而是由操作系统控制。从应用程序的角度看，操作系统把CPU时间分派给线程，所以线程并不是以资源的方式获取CPU。

资源常常依赖于其他资源。比如，文件句柄是一种资源，代表了文件，文件又可被看做另一种资源。另一个例子是服务由组件提供，而组件会包含线程和内存资源。这些资源的依赖关系可以用跨越多个层次的资源提供者的图来表示。

下面的图1-1概括了资源的类型，并为每类举了一个例子。

	可重用	不可重用
独占式	内存	处理时间
并行式	只读对象	-

图 1-1

1.1 资源管理概览

软件系统中的资源管理是控制资源对于资源使用者可获取性的过程。资源使用者可以是任何获取、访问、释放资源的实体。资源管理的内容包括：确保资源在需要时可获得，确保资源的生命周期是确定的，确保它们被及时释放以保证使用它们的系统不会被拖垮。

管理资源并不容易，有效地管理它们更不容易。软件的非功能需求，比如性能、伸缩性、灵活性、稳定性、安全性和服务质量常常非常依赖于高效的资源管理。这些非功能需求是影响软件如何设计和实现的作用力（force）[⊖]。虽然单个地看，开发系统时每个作用力都可应付，但要在这些力之间寻求平衡却极具挑战性。为了达到平衡，我们需要先弄清楚一些问题。比如，系统的性能是否比让系统变得灵活和易于维护更重要？系统响应时间的可预测性是否比它的可伸缩性更重要？服务的第一次访问时间重要还是平均访问时间重要？

同时应对多个作用力很具挑战性。这是因为应对其中之一常常需要对系统的其他方面加以妥协。比如，灵活性常常需要以牺牲系统性能的代价来获得。类似地，为特定用例（比如对服

⊖ 侯捷先生在《内存受限系统之软件开发》（《Small Memory Software》中译本）中对“force”一词采取保留原文不译的做法。该书第291～309页对“force”有详尽阐述。本书则视情况或保留原文或译做“作用力”。——译者注

务的第一次访问）进行优化通常会导致复杂性的增加，还经常会导致处理一般用例时延迟的增加。与应付这些常彼此相反的作用力带来的挑战相伴的，是人们对高效资源管理的强烈依赖。应付大多数作用力通常意味着改变获取资源、访问资源和管理资源的方式。

设计具有高效资源管理的系统时需要考虑下面的主要作用力：

- 性能。性能至关重要的系统具有很多属性，包括低延迟和高吞吐量。因为每个动作通常都会涉及很多资源，所以避免不必要且会带来处理开销和延迟的资源获取、释放及访问非常重要。
- 可伸缩性。大且复杂的系统通常会有很多资源使用者，需要多次访问资源。在很多情况下，系统设计时就考虑了用例，比如预期的使用者数目。这些用例经常随着新需求的增加而不断扩展。例如，新的需求可能会要求支持更多的用户、更高的传输量，而要求对系统性能影响尽可能小。如果系统能满足这些需求，我们就说系统是可伸缩的。要实现这个目标，管理资源和管理它们生命周期的方式扮演着重要角色。除了同步开销之外，资源的获取和释放占用的CPU周期百分比是最多的[SMFG01]。除了伸，缩也很重要。这就需要为大系统设计和开发的软件也能适应小系统，以较低的资源开销运行。
- 可预测性。可预测性意味着系统行为同预期的一样。在具有实时需求的系统中，单个操作的最大响应时间必须是可预测的。为了获得可预测性，系统必须小心地管理资源。当进行优化时，不能为了性能或者可伸缩性而牺牲服务的可预测性。
- 灵活性。系统的一项常见需求是易于配置和自定义。这意味着在编译、初始化或者运行时改变系统配置应该是可行的。真正灵活的系统给予了用户自由。所以，就资源管理而言，获取、释放资源的机制以及资源生命周期的控制应当具有灵活性，但还要满足性能、可靠性和可伸缩性需求。
- 稳定性。软件系统的一项重要需求是经常获取和释放资源不应使系统变得不稳定。资源分配和管理的方式必须能带来对资源的有效利用，并且能避免可能导致系统不稳定的饥饿状态的出现。如果多个资源相互交互，或者作为整体影响某个动作，那么必须确保系统不会因某个资源的失败而进入不一致的状态。
- 一致性。所有的资源获取、访问、释放操作之后软件系统必须仍然处于一致的状态。特别是必须很好地管理资源之间的相互依赖关系，以确保系统的一致性。

因此，如我们所见，这些作用力大多数都相互影响，所以要想应付某种作用力而不影响其余是很难的。所以，资源管理才那么困难，以组织良好的方式解决这些问题才如此重要。

1.2　资源管理的范围

在几乎所有的领域中资源管理都是一个重要方面，不管是嵌入式应用程序[NoWe00]还是大型企业系统[Fowl02]抑或是网格计算[BBL02]都如此。任何系统都可从高效的资源管理中受益，不管资源是何类型，资源可获得性如何。例如，为小的嵌入式设备设计的应用程序通常具有很有限的资源，所以对良好资源管理的需求显而易见。最主要的限制因素是CPU性能、可使用的内存以及网络和总线的有限带宽。典型的嵌入式应用程序包括移动电话和手持设备的软件。

类似地，大型企业系统可能运行电信交换或者电子商务应用程序。这样的系统上的应用程序通常会基于框架和服务器端组件系统来构建。在这样的系统中，高效的资源管理对于确保可伸缩性至关重要，因为为了降低成本，应避免增加额外硬件。

在开发阶段的晚期（系统集成和测试阶段，进行系统和性能分析的时候）才发现资源管理需求的情形也并非少见。但是，在这个时候改变系统设计很困难，而且代价高昂。因此，在软件系统的生命周期中，尽早应对资源管理的挑战很重要。

资源是从某种资源提供者获得的。例如，内存是从操作系统获得的。因为获取资源通常是代价昂贵的操作，所以必须正确地控制获得的资源的生命周期，以避免额外的开销。因为不使用的资源也可看做是开销，所以记得及时释放也很重要。

处理影响资源获取、访问和释放的主要作用力很重要，寻找简单的解决方案同样重要。带来的开销比好处多的解决方案没有用处。高效资源管理应当为应用程序开发者带来简单的编程模型。例如，为资源管理的方式引入一层透明性有助于简化开发者的工作。

大多数现代应用程序都以层次[POSA1]构架的形式建立。典型情况下，操作系统的抽象层形成了最低层，然后是中间件层[POSA2][VKZ04]、组件或者基本服务层[VSW02]，最后是实际应用程序逻辑层（见图1-2）。

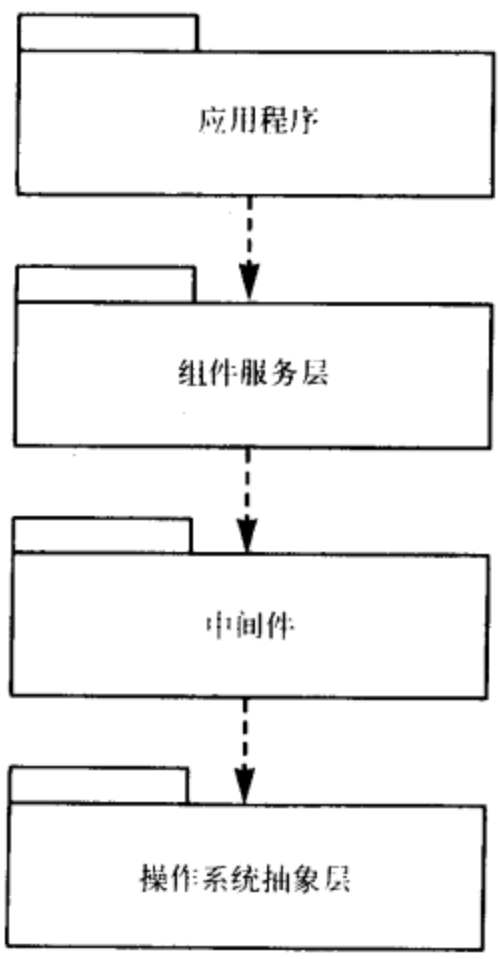


图 1-2

资源管理并不限于其中任何一层。相反，它适用于所有的抽象层次。例如，操作系统的资源（文件句柄和内存之类）需要在操作系统抽象层小心处理。类似地，中间件曾需要管理很多类型的资源，比如连接和线程。

资源管理不仅对于本地系统重要，对于分布式系统也很重要。分布式系统管理起来要复杂很多，因为远程通信是不可靠的。结果就是，管理跨进程边界分布的资源要困难很多。

1.3 模式的使用

几乎所有的领域中，模式都有助于表示解决问题的最佳实践。模式已经应用到了很多领域，从建筑结构[Alex79]到软件系统的构架再到教学[Peda04]。在本书中，我们专门研究用于软件构架的模式。

软件构架中的模式能够表示在一个具体的情形下如何使用一个或多个设计原则来找到最佳的解决方案。如[POSA1]中描述的，设计原则的应用通过很多支持手段（比如抽象、封装和模块化）来体现。使用模式有助于识别并记录一部分超出单个类和实例层次的设计原则。这些识别出的抽象有助于对成功软件构架和设计的重用。通过重用成功的软件构架，可以避免软件设计中常见的错误和陷阱。若想对“模式是什么”以及模式的历史获得更深入的理解，请参考[POSA1]和[POSA2]。

如前所述，高效的资源管理无可避免地对任何种类软件的设计和开发都会有影响。在多层系统中，资源管理对系统的每层甚至跨层都具重要性。资源管理从系统的角度正确实现是至关重要的。

在系统中实现资源管理时必须用到的设计技法很大程度上取决于领域、系统约束和系统需求。模式抽象自特定的领域，并且受系统需求和约束驱使。所以，它们提供了描述“适合用于特定系统的设计手法”的最有效的方式。模式的基本要素之一就是其所适用的环境。这可以用于辨识在系统中是否应当使用某种特定的资源管理策略。

模式的另一个关键要素是它有助于解决的作用力的集合。在资源管理环境中，这能够用于辨识系统中能够应用某种模式的部分。

1.4 资源管理中的模式

在本章的开始部分，我们列出了对应于典型软件系统的一组非功能需求的作用力。因为它们会影响资源管理，所以解决它们可以为软件系统带来很大的好处。本书展示了一种用于资源管理的模式语言，可以解决这些作用力。在本书中，资源包含了本章前面部分定义的所有东西。书中展示的每个模式都是独立完整的，但它们共同形成了一种内聚的模式语言，可以帮助软件开发者和设计者应付一部分资源管理的挑战。下面是本书中列出的模式以及它们解决的作用力的提纲：

- 性能。Eager Acquisition模式有助于加快第一次资源访问，从而加快整个系统的响应速度。Caching模式通过避免对常用资源的昂贵的重复获取操作来提升性能。Pooling模式减少了开销很大的用于释放和重新获取资源的操作，从而提升了性能。
- 可伸缩性。Leasing模式和Evictor模式可帮助释放不用的资源，降低资源饥饿的风险，从而提高性能的可伸缩性和稳定性。Coordinator模式提供了一种解决方案，随着系统参与者数目变化而具可伸缩性。Caching模式和Pooling模式有助于避免昂贵的资源获取和释放，从而有

助于提升系统的可伸缩性。类似地，Lazy Acquisition模式和Partial Acquisition模式则确保资源只在实际需要时才被获取，从而降低了在特定时间点需要资源的数目。Resource Lifecycle Manager模式通过管理资源的生命周期并确保维护优化数目的资源的方式来支持可伸缩性。

- 可预测性。Eager Acquisition模式避免了在运行时执行昂贵的资源获取操作。它通过确保资源获取和访问的响应时间很短而增加了系统的可预测性。同理可知，Pooling模式也提升了系统性能，因为资源是从资源池中获取的。
- 灵活性。Lookup模式把资源提供者同资源使用者解耦合了。这一解耦合使得系统整体变得更为灵活。Partial Acquisition模式和Lazy Acquisition模式则在另一个层面上带来了灵活性：我们拥有替换获取策略的灵活性。

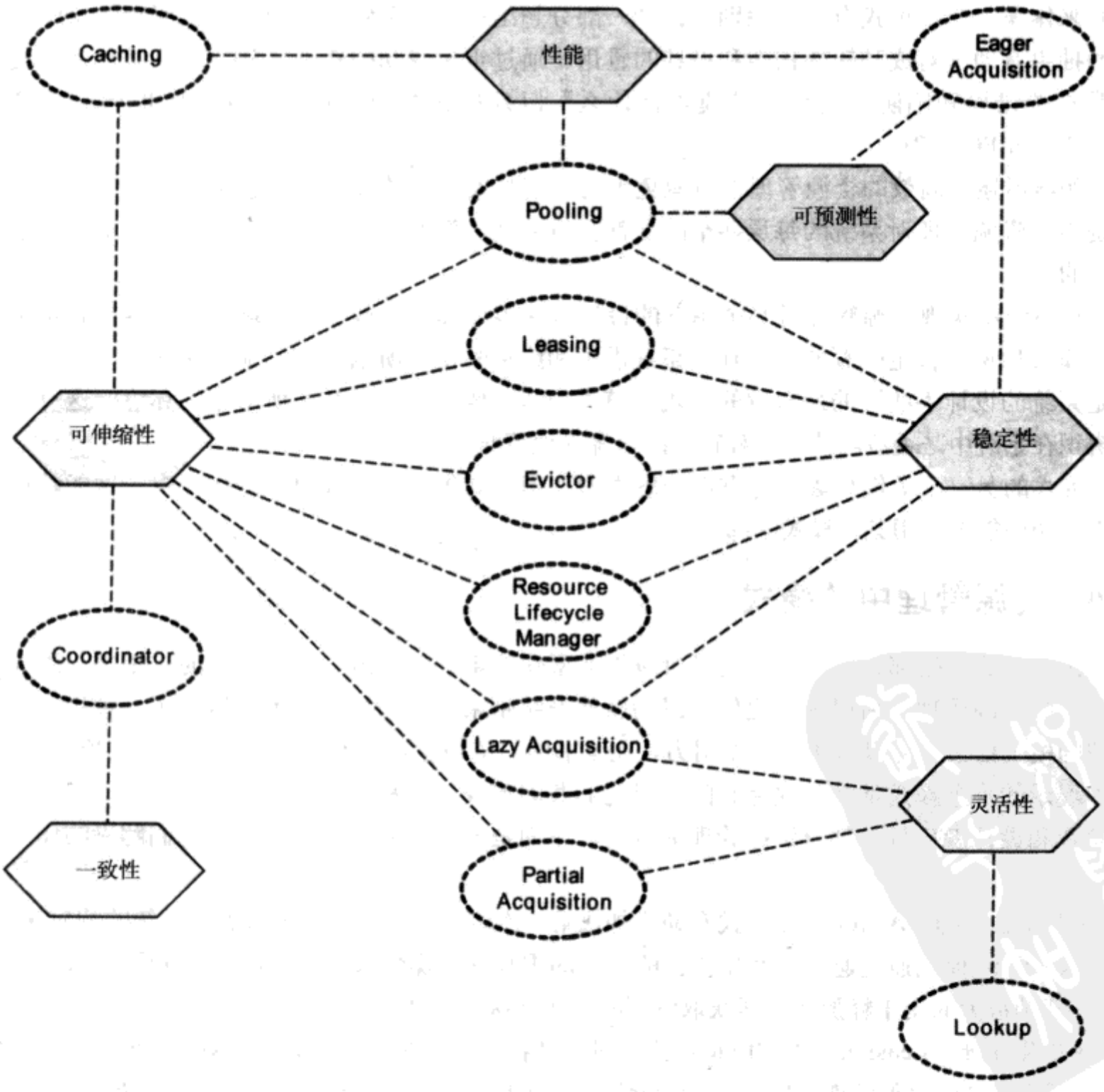


图 1-3

- 稳定性。Lazy Acquisition模式和Eager Acquisition模式减少了任意给定时间的资源消费。这些模式有助于尽量减少资源的消费，确保系统的稳定性。Pooling模式则通过避免经常执行低层次的获取资源及释放资源操作而提升了系统的稳定性。Evictor模式则降低了资源耗尽的可能性，从而也增强了应用程序的稳定性。Leasing模式减少了对未用的资源的消耗并避免了资源使用者对非法资源的访问，从而有助于提升系统的可靠性和稳定性。Resource Lifecycle Manager模式可确保仅当有足够资源可用时才会把资源分配给用户。这使得系统更稳定，因为避免了用户通过直接从系统获得资源而可能导致的资源饥饿情形。
- 一致性。Coordinator模式有助于保证本地以及分布资源间的一致性。

请注意，前面所述的模式是按照它们解决的作用力而划分的。而详细描述这些模式的章节则是按照资源管理的不同阶段来划分的。具体地说，分为资源获取、资源生命周期、资源释放阶段。当我们进行案例讨论时会重新基于作用力来划分模式。

图1-3展示了非功能性需求和本书展示的资源管理模式之间的关系。

1.5 相关工作

软件构架模式不是孤岛，而是同其周边环境[⊖]相互连接的。每个模式都会有个初始环境和终止环境。一个模式的终止环境会形成另一个模式的初始环境。所以描述一个模式常常需要提及好几个其他模式。当时解决一个特定问题时，可以认为这些被提及的模式同被描述的模式是相互紧密配合的。从另一个角度来看，模式可以被归结为模式系统[POSA1]，这些系统描述了软件开发的特定问题域。[⊖]

通常而言，模式并不仅限于一个行业（比如电信业），也不会只是用户的一个领域(比如嵌入式系统)。模式可以用于很多共享相同核心情景（模式的环境）的地方。

软件设计者和开发者对模式的广泛兴趣发源自一本书，即Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides（他们4人常被称做“Gang of Four”）写的《Design Patterns: Elements of Reusable Object-Oriented Software》[GoF95]。他们的书描述了使用面向对象技术和框架来开发灵活软件的最佳实践。一年之后，作为POSA系列中的第一本，《Pattern Oriented Software Architecture - A System of Patterns》[POSA1]出版了。这本书辑录了基本的构架模式，并为设计和构建软件框架及应用程序提供了建议。GoF95和POSA1都围绕软件构架模式展开，但都未涉及资源管理的领域。

随着模式工作的开展，Hillside Group[Hill04]组织的PLoP和EuroPLoP之类的会议提供了在社区中分享新模式的活跃的论坛。Pattern Language of Program Design （PLoPD）系列[PLoPD1][PLoPD2][PLoPD3][PLoPD4]概括了发表的材料。若想知道截至2000年所发表的所有

⊖ context可译为环境、情景、背景、上下文等。本书中一般译为环境、背景，有时也会出于语意连贯性考虑译为情景。——译者注

⊖ 微软公司Patterns & Practices部门的Ward Cunningham主导的http://patternshare.org网站以改进自C2 wiki的形式提供了按照问题域划分的模式表格。——译者注

模式的完整列表，可参考Linda Rising写的《The Pattern Almanac》[Risi00]。

现有的专注于特定问题领域的模式著作对于资源管理的话题只是打“擦边球”而已。下面列出了一些最重要的著作。若想得知每个相关模式，请参考本书模式中的“又见（See Also）”小节。

- 同步和网络对象。《Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects》[POSA2]记录了同步及面向网络的应用程序以及中间件中常出现的模式。书中包含的同步模式，比如Active Object、Monitor Object、Half-Sync/Half-Async、Leader/Followers、Thread-Specific Storage，描述了如何有效地处理对资源（比如对象、事件源）的同步访问。因为POSA2对同步模式的领域覆盖广泛，所以我们在本书中就不再提及资源管理的同步问题。另一本针对Java讲述同步模式的好书是Doug Lea的《Concurrent Network Programming》[Lea99]。对于针对实时系统的模式，我推荐Bruce P. Douglass的《Doing Hard Time》[Doug02]。
- 远程通信。《Remoting Patterns》[VKZ04]中描述的模式语言在POSA2的基础上更进一步，它描述了如何建立分布式对象基础设施。它描述了通用中间件技术（比如CORBA OMG04a]、Web Service[W3C04]、.NET Remoting[Ramm02]）的组成部分。这些分布式对象基础设施管理的远程对象是资源。因此，本书中的一些模式，比如Lookup模式、Lazy Acquisition模式、Pooling模式和Leasing模式在那个模式语言中也反复出现，但仅限于远程对象。
- 组件基础设施。《Server Component Patterns》[VSW02]包含了一个用于今天的组件基础设施（比如EJB[Sun04b]和COM+[Ewal01]）的模式语言。这些模式发展了POSA2和《Remoting Patterns》[VKZ04]中描述的模式。组件是由组件基础设施管理的资源。所以《Server Component Patterns》包含了Naming模式和Instance Pooling模式，它们是我们的模式语言中Lookup模式和Pooling模式的特化。此外，Managed Resource模式是我们的模式语言中Resource Lifecycle Manager模式的应用。
- 容错。容错性有助于确保系统的一致性和稳定性，因此同资源管理模式相关。容错技术（比如[Lapr85]和[ALR01]中描述的那些）在今天的高可靠性系统中很常见。Titos Saridakis[Sari02]以模式的形式覆盖了这些技术。他的模式可以分为错误检测、恢复和屏蔽这3类。这3类中，单个的模式可以组合起来以建立特定的容错解决方案。容错技术通过检测出失败或者出错的资源并提供从这类错误中恢复或者屏蔽错误的手段来确保系统的稳定性。
- 内存有限的小型系统。Charles Weir和James Noble在《Small Memory Software》[NoWe00]一书中描述了用于在小型/嵌入式系统中有效管理内存的模式。因为内存是资源管理中的典型资源，所以一些模式是类似的。例如，[NoWe00]中的Paging模式是Caching模式的特化，Variable Allocation、Fixed Allocation和Pooled Allocation分别是Lazy Acquisition、Eager Acquisition和Pooling的特化。Garbage Collection模式[JoLi96]同Evictor模式相关。重叠是很自然的，并没有多余，因为[NoWe00]给出了如何将这些概念用于内存的详细指导，而本

书则给出了更一般化的指导。

- 企业系统。Martin Fowler的《Patterns of Enterprise Application Architecture》[Fowl02]展示了在今天的企业系统中最常用的模式。对企业系统而言，性能和可伸缩性的作用力也很重要。因此，书中除了描述很多其他模式，也描述了两个有助于应对这些作用力的模式：Lazy Load模式和Data Transfer Object模式，它们分别同Lazy Acquisition模式和Caching模式相关。

另一本很好地补充了Fowler的著作的书是《Enterprise Integration Patterns》[HoWo03]。这本书中的模式描述了在企业应用中如何设计、创建和部署消息传递解决方案。

《Enterprise Solution Patterns Using Microsoft .NET》[TMQH+03]则记录了用.NET技术[Rich02]构建企业系统的最佳实践。[TMQH+03]中记录的Page Cache模式同Caching模式直接相关。

另一本关于企业解决方案的好书是Paul Dyson和Andrew Longshaw写的《Architecting Enterprise Solutions: Patterns for High-Capability Internet-based Systems》[DyLo04]。作者给出了如何构建大规模企业解决方案的构架和实践指导。这些模式和它们解决的系统的非功能性特性之间的联系非常有价值。他们的模式Resource Pooling和Local Cache同本书中的模式直接相关。

只以特定的应用领域为背景记录模式不必要地把模式的可用性限制为那个领域。在很多情况下，应用领域可以彼此借鉴。例如，实时和嵌入式领域中的软件开发项目正越来越多地广泛借鉴企业领域中的最佳实践和技术。这已经被证实是成功的，因为实时领域正从今天更高性能的CPU可带来的更高的抽象层次中获益。在过去的几年里，OOPSLA（Object-Oriented Programming Systems, Languages, and Applications）等会议上多次关于分布式实时和嵌入式系统的模式的研习和讨论证实了这一点。

1.6　模式格式

本书中的所有模式都是独立完整的，并遵守POSA1的模式格式。这种格式使得我们既可表现出模式的本质，又可描述关键细节。我们的目标是同时满足想要看单个模式的基本理念概要的读者和想详细知道模式如何共同工作的读者。

POSA格式的每一节都为下一节搭筑起“舞台”。“实例（Example）”这一节引出了“背景（Context）”、“问题（Problem）”和“解决方案（Solution）”，它们概述了模式的本质。“解决方案（Solution）”这一节引出了“结构（Structure）”和“动态（Dynamics）”这两节，它们展示了关于模式如何工作的更详细的信息，为读者阅读后面的“实现（Implementation）”一节做好准备。

“实例解析（Example Resolved）”、“变体（Variants）”、“已知应用（Known Uses）”、“结果（Consequences）”和“又见（See Also）”这些小节把模式的描述补充完整。我们包含了很多交叉引用，以帮助读者理解本书中的模式以及其他发表的模式之间的关系。

每个模式的“实现（Implementation）”这一节提供了详细的步骤，你可根据这些步骤来实现模式。这一节还提供了示例代码。如果你想先概览所有模式，那么你可能会想在读第一遍时跳过“实现（Implementation）”小节，当你需要知道某个特定模式的实现细节时再回过头来读这些小节。至于用来解释我们的模式的结构和行为的图表，我们尽可能地遵照标准

UML[Fowl03]来绘制。

模式的“结构（Structure）”一节使用了CRC卡片[BeCu89]。CRC卡片有助于以一种非正式的方式辨别和说明应用程序的对象或组件。特别是在软件开发的早期阶段CRC卡片尤其有用。它可以描述组件、对象或者一类对象。卡片包含3个字段，描述组件名、职责，以及其他协作组件的名字。

模式的“变体（Variants）”一节描述了从该模式派生出来（扩展或改变问题和/或解决方案）的相关模式。

模式的“特化（Specialization）”一节则记录了如何使用模式的一种特化形式来解决特定的问题或情形。因为模式通常会把这样的细节抽象掉，所以这一节有助于描述可应用模式特化形式的情形。

第2章 资源获取

"我发现，我获得的知识很大一部分是在寻找其他信息的过程中得到的。"

——Franklin P. Adams

概览

资源的生命周期始于获取。资源如何以及何时被获取，对于软件系统的运行会有重要影响。通过优化获取资源所需的时间，系统性能可以显著改善。

在资源可以被获取之前，要解决的最基本问题是如何发现资源。Lookup模式解决了这个问题，并且描述了如何通过资源提供者让资源可获得，以及资源使用者如何发现这些资源。

当资源被发现之后，就可以被获取。资源获取的时间很重要，Lazy Acquisition模式和Eager Acquisition模式解决了这个问题。Lazy Acquisition把资源的获取推迟到可能的最晚时分，而Eager Acquisition则尽可能早地努力获取资源。两个极端都很重要，它们非常依赖于资源的用途。

如果被获取的资源不是立即被使用，那么可能会导致浪费。Lazy Acquisition解决了这个问题，从而会带来更好的系统伸缩性。而另一方面，有些系统有实时限制，对资源获取时间有严格的需求。Eager Acquisition解决了这一问题，从而会带来更好的系统性能和可预测性。

我们可能并不需要整个的大型资源或者未知大小的资源，因此可能获取资源的一部分就够了。Partial Acquisition模式描述了如何把资源获取分割成一个个步骤，使得我们可以只获取资源的一部分。Partial Acquisition可以用做Lazy Acquisition和Eager Acquisition之间的桥梁。Partial Acquisition的第一步通常是积极地（eagerly）获取资源的一部分，后续的步骤则把进一步的资源获取推迟到较晚的时候。

Lookup模式既可用于可重用的资源，也可用于不可重用的资源。此外，它既可支持可以同步访问的资源，也可支持独占式资源。这是因为Lookup模式通常仅仅关注提供对资源的访问，却并不关心资源实际上如何被使用。

Lazy Acquisition、Eager Acquisition和Partial Acquisition也都既可用于可重用的资源，也可用于不可重用的资源。但是，如果一个可重用的资源是可同步访问的，那么这3个模式可以提供一定程度的优化，使得一个资源只被获取一次，而被多个用户同步使用。这样的优化要求在Partial Acquisition的情况下要求特别处理，这是因为部分获取的资源量可能对于不同的同步用户数有所不同。

2.1 Lookup模式

Lookup（查找）模式描述了如何通过使用查找服务作为中介实例来发现和访问资源（不管资源是本地的还是分布的）。

实例（Example）

思考一下一个通过CORBA[OMG04a]实现为一组远程对象的几个服务组成的系统。为了访问这些分布式服务中的一个，客户通常需要获取提供服务的对象的引用。对象引用标识了将会获得请求的远程对象。对象引用可以作为操作的参数或者请求的结果在系统中传递。所以客户可以从系统中的一个对象获得另一个远程对象的引用（见图2-1）。但是，客户如何获得想要访问的对象的第一个引用呢？

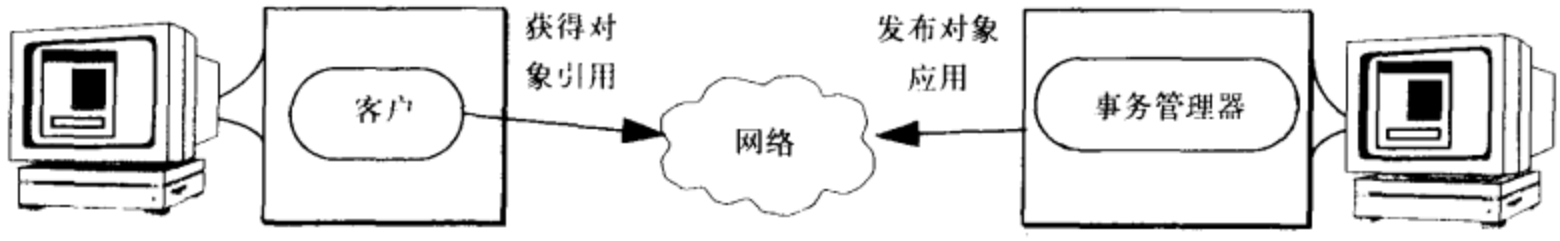


图 2-1

比如，在提供分布式事务服务的系统中，客户可能想要获取事务管理器的引用。服务器如何建立事务管理器对象的引用，才能使得它可被广泛获取呢？客户如何在没有对其他对象的引用的情况下获得事务管理器对象的引用呢？

背景（Context）

资源使用者需要寻找并访问本地和分布式资源的系统。

问题（Problem）

资源提供者可能提供了一个或多个资源给资源使用者。随着时间的流逝，额外的资源可能会增加出来，或者存在的资源可能被资源提供者移走。资源管理者发布存在资源的可获取性的一种方式是以一定的时间间隔向感兴趣的资源使用者发布广播消息。这样的消息必须每隔一段时间就发布一次，以确保加入系统的新的资源使用者知道可用的资源。相反，资源使用者也可发送广播消息请求所有可用的资源提供者响应。一旦资源使用者从所有可用资源提供者处获得了回复，它就可以选择其所需要的资源。但是，两种方式可能都相当低效，并且代价高昂，因为它们产生了大量的消息，若是分布式系统的话，这些消息会充满整个网络。为了解决问题，让资源提供者发布资源，并且让资源使用者以开销较小的方式有效地找到这些资源，需要解决如下作用力：

- 可用性（Availability）。资源使用者应当可以根据需要找到在其环境中有哪些资源可用。
- 自举（Bootstrapping）。资源使用者应该可以获取资源提供者的初始引用。
- 位置独立性（Location independence）。资源使用者应当可以从资源提供者获取资源，无论资源提供者位于何处。类似地，资源提供者应该可以为资源使用者提供资源，而不必知道资源使用者位于何方。
- 简单性（Simplicity）。当查找资源时，解决方案不应当给资源使用者带来负担。类似地，解决方案也不应当为资源提供者带来负担。

解决方案（Solution）

提供资源查找服务，使得资源可被资源使用者获得。资源提供者通过查找服务来发布资源，

并附有描述其发布的资源的属性。这样资源使用者就可以先通过属性来查找被发布的资源，然后获取资源，再使用资源。

对于需要在使用前获取的资源，资源提供者把指向它们自己的引用同描述其提供的资源的属性一起注册，这样资源使用者就可以获得这些被注册的引用，并使用它们来从被引用的资源提供者获取可用资源。

对于不需要在使用前获取而可以直接访问的资源，比如可同步重用的服务，资源提供者把指向资源的引用同描述资源的属性一起注册。这样资源使用者就可以直接访问资源，而不必先同资源提供者打交道。

查找服务成为了资源使用者和资源提供者之间的通信中点。它使得资源使用者在明确需要从资源提供者获取资源时可以访问资源提供者。此外，查找服务也使得资源使用者可以直接访问不需要从资源提供者获取的资源。在这两种情况下，资源使用者都不需要资源提供者的位置。类似地，资源提供者也不需要知道想要获取并访问它们提供的资源的资源使用者的位置。

结构（Structure）

下列参与者形成了Lookup模式的结构:

- 资源使用者（Resource User）使用资源。
- 资源（Resource）是一个实体，比如一个提供某种类型的功能的服务。
- 资源提供者（Resource Provider）提供资源，并通过查找服务发布资源。
- 查找服务（Lookup Service）提供了让资源提供者通过到其自身的引用发布资源，并让资源使用者找到这些引用的能力。

下面的CRC卡片描述了这些参与者的职责（Responsibility）与协作（Collaboration），如图2-2所示。

Class Resource User	Collaborator
Responsibility • 寻找资源 • 使用资源	• Resource • Lookup Service • Resource Provider

Class Resource	Collaborator
Responsibility • 提供某类功能	

Class Resource Provider	Collaborator
Responsibility • 提供资源给资源使用者 • 通过查找服务发布资源及其属性	• Resource • Lookup Service

Class Lookup Service	Collaborator
Responsibility • 允许资源提供者发布资源 • 允许资源使用者查找发布的资源 • 将属性同资源相关联	

图 2-2

下面的类图描述了Lookup模式的结构（见图2-3）。

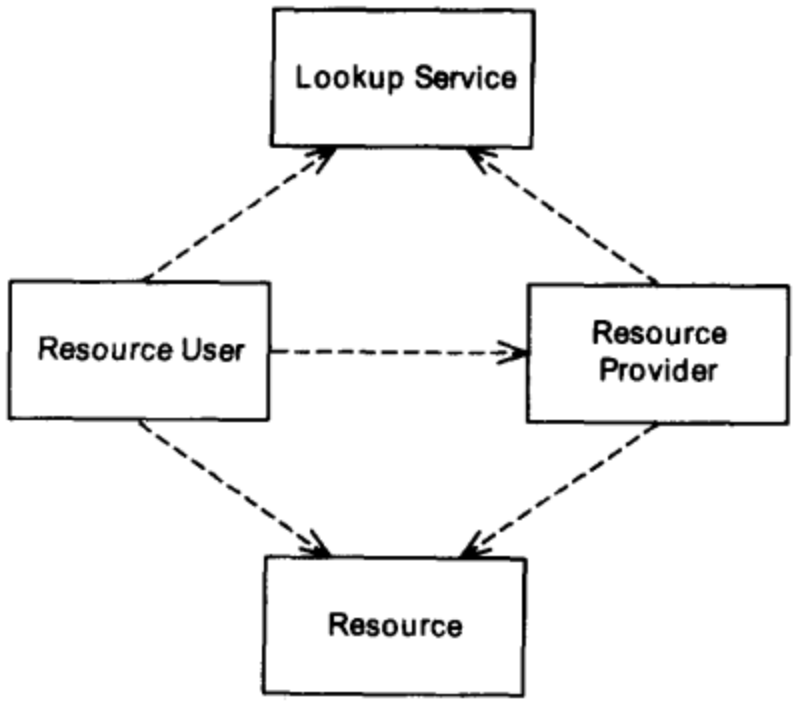


图 2-3

资源使用者依赖于所有的其他3个参与者：依赖查找服务来寻找资源；依赖资源提供者来获取资源；依赖资源来实际访问它。

动态（Dynamics）

Lookup模式有3组交互场景。

场景1：在第1个交互场景中，资源提供者通过查找服务发布资源。默认资源提供者已经知道查找服务的访问点。若想知道资源提供者不知道查找服务的访问点时的交互，请参考场景3。当发布资源时，资源提供者将与描述其提供的资源类型的属性（property）一起，向查找服务注册（register）一个到它自身的引用（reference），如图2-4所示。

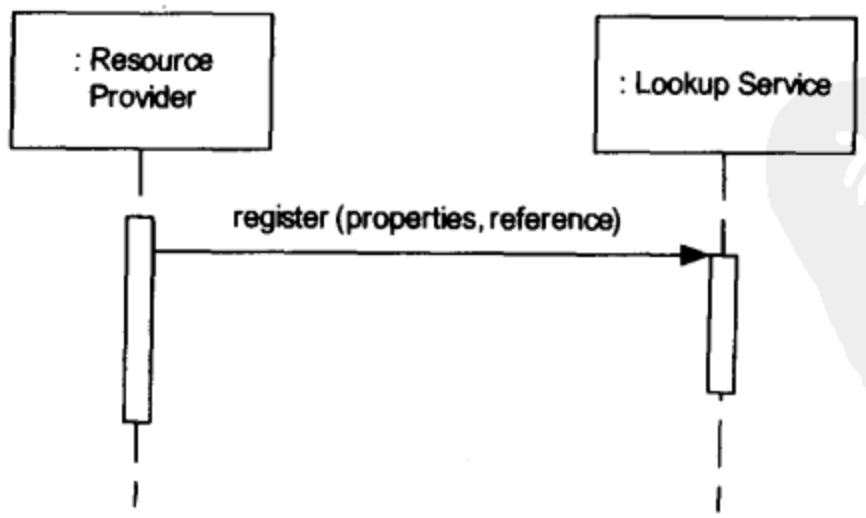


图 2-4

场景2：在第2个交互场景中，资源使用者通过查找服务找到了一个资源提供者，包含了如下交互（见图2-5）：

- 资源使用者借助查找服务，通过一个或多个属性（比如资源描述、接口类型、位置）来寻

找需要的资源。

- 查找服务响应并返回能提供所需资源的资源提供者的引用。
- 资源使用者利用资源提供者的引用来获取（acquire）并访问（access）资源。

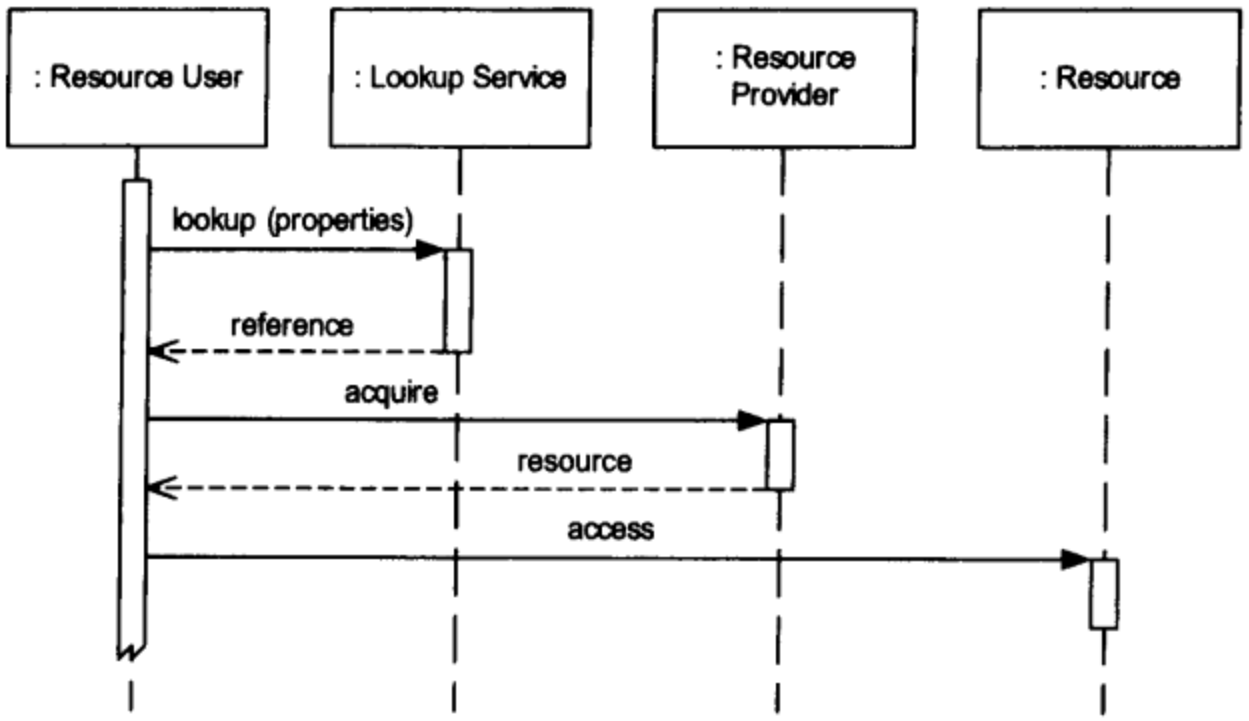


图 2-5

场景3：在分布式系统中，资源使用者和资源提供者可能并不知道查找服务的访问点。在这样的情形下，访问点可以通过应用程序的运行时环境来配置，或者应用程序也可以使用一个自举协议来发现访问点。自举协议可以是广播、多播或者几个单播消息的组合。

所需的步骤有:

- 资源提供者或者资源使用者通过自举协议寻找查找服务。
- 查找服务响应并声明它的访问点（access point）。

下面的顺序图显示了这些交互，它们对于资源提供者和资源使用者都是正确的（见图2-6）。在图表中，资源提供者使用广播（broadcast）作为自举协议。

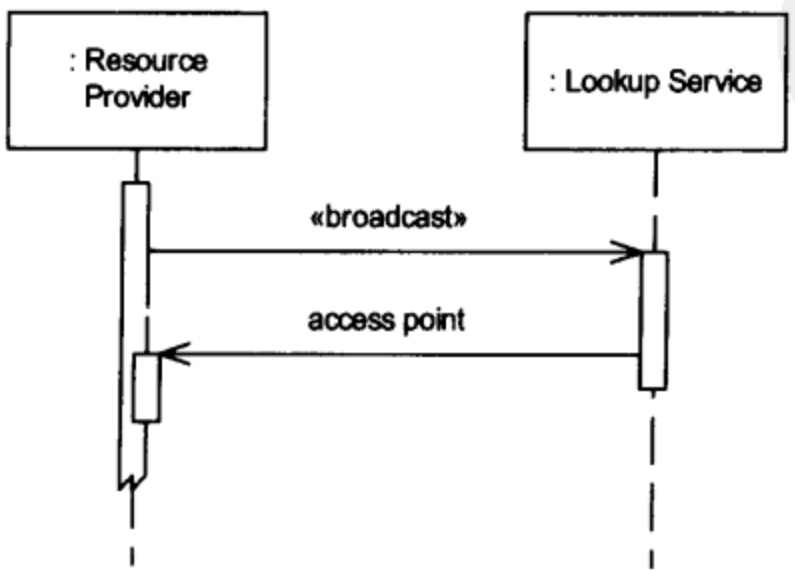


图 2-6

实现（Implementation）

实现Lookup模式有五步：

1）*决定查找服务的接口*。查找服务应当为资源的发布和查找提供方便，或者直接实现，或者通过它们的资源提供者来实现。它应当提供一个接口，让资源提供者可以注册以及注销引用。引用与描述所提供的资源的属性相关联。查找服务保存有一个注册引用以及相关联属性的列表。这些属性可以被查找服务用于基于资源使用者发送的查询来选择一个或多个资源。在最简单的情形下，属性可以只包含资源提供者提供的一个资源的名称或者类型。资源使用者通过查找服务执行查询获得的结果或者是合法的引用，或者是错误代码（如果无法找到匹配的资源的话）。

可以为查找服务定义不同的策略。例如，查找服务可以支持与重复的名称或者属性的绑定。查找服务应当提供一个接口，让资源使用者可以获取所有可用资源提供者的列表。资源使用者用到的查找条件可以仅仅是通过名字查找，也可以是如第5步“决定查询语言”所描述的更复杂的查询机制。

2）*决定是注册资源引用还是资源提供者的引用*。取决于资源类型和获取策略，可以注册资源引用或者资源提供者的引用。如果资源必须先由资源使用者显式获取，那么就应当向查找服务注册提供那个资源的资源提供者的引用，并且一起注册描述资源提供者所提供的资源的属性。要求显式获取资源有利于控制可以访问资源的资源使用者的类型。

当资源提供者提供多个资源时，资源提供者的引用可以用于辨识资源。这有助于允许资源提供者把获取请求同需要的资源相关联。更多细节请参见Multi-faceted Resource Provider变体。

另一方面，如果资源不需要被显式获取，而是可以被资源使用者直接访问，那么就可以向查找服务注册资源的引用，并且一起注册描述资源的属性。例如，对于可以同时被重用的资源，可以通过向查找服务注册资源引用的方式来使得资源使用者可以直接使用资源。这类资源的例子包括只读对象和Web Service [W3C04]。这样的资源或者没有同步问题，或者自身可以对访问同步。

3）*实现查找服务*。在内部，查找服务可以用很多种方式实现。比如，查找服务可以用某种树型数据结构来保存注册的引用以及它们的元数据，这在必须为复杂的依赖关系建模时很有帮助，或者也可以用简单的哈希表来保存。对于非关键并且经常改变的资源发布，信息可以临时性地存储；而对于关键的发布，则应该把关联以某种恰当的后端持久化机制来持久存储。

比如，一种CORBA实现Orbix [Iona04]就用COS Persistent State Service来把到它的Naming Service的名称绑定持久化，这是查找服务的一种实现。其他的CORBA实现，比如TAO [Schm98] [OCI04]则用内存映射文件来把绑定持久化。

4）*提供查找服务访问点*。为了同查找服务通信，访问点是必需的。访问点可以是一个Java引用、一个C++指针或者引用，也可以是一个通常会包含查找服务所在的主机名、端口号的分布式对象引用，等等。这一信息可以以几种方式发布给资源提供者和资源使用者，比如写到一个资源提供者和资源使用者可以访问的文件，或者通过定义好的环境变量。

例如，很多CORBA实现都用客户可以访问的属性或者配置文件来发布Naming Service的访问点。

如果查找服务没有发布访问点，那么就需要设计一个自举协议来使得资源提供者和资源使

用者可以获得访问点。这样的自举协议通常会用广播或者多播协议来设计。资源提供者或者资源使用者用自举协议发送一个要求查找服务的引用的初始请求。收到请求之后，通常一个或者多个查找服务会向请求者发送回应，把它们的访问点传递过去。然后资源提供者就可以同查找服务联系以发布自己的引用。类似地，资源使用者可以同查找服务联系以获得注册的资源提供者的引用。

在CORBA中，客户或者服务器可以对ORB执行resolve_initial_references()调用来获得Naming Service的访问点（比如，一个对象引用）。

5) 决定查询语言。查找服务可以选择支持一种查询语言，使得资源使用者可以用复杂的查询来寻找资源。例如，查询语言可以基于描述资源使用者感兴趣的资源类型的属性表。当使用查找服务来查询资源时，资源使用者可以提交一个被请求资源必须满足的属性列表。查找服务可以比较资源使用者提交的属性列表和可用资源的属性。如果匹配，那么就把相应资源或者资源提供者的引用返回给资源使用者。

CORBA Trading Service [OMG04f]是一项查找服务，允许说明被注册资源相对应的属性。资源提供者是向查找服务注册CORBA对象的服务器应用程序。对象引用将指向服务器应用程序，但也会把CORBA对象标记为资源。作为资源使用者的客户可以使用同被注册的CORBA对象相匹配的标准来建立任意复杂的查询。到服务器应用程序中的CORBA对象的引用会被返回给客户。

实例解析（Example Resolved）

思考下面的例子，客户想要获取一个分布式CORBA环境中的事务管理器的初始引用。使用Lookup模式，需要实现一个查找服务。大多数CORBA实现都提供了这样的查找服务，这些服务或者是以Naming Service的形式，或者是以Trading Service的形式，或者两者兼备。这些服务可以通过IIOP协议（Internet Inter-ORB Protocol）访问，它们提供了精确定义的CORBA接口。

在我们的例子中，创建事务管理器的服务器是资源提供者。服务器应当首先获得Naming Service的引用，然后使用这个引用来注册建立的事务管理器的引用。引用包含了服务器的访问点，并标明了被注册的服务器中的资源。下面的C++代码展示了服务器如何获得Naming Service的引用并用它来注册事务管理器。

```
// First initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Create a transaction manager
TransactionMgr_Impl *transactionMgrServant =
   new TransactionMgr_Impl;

// Get the CORBA object reference of it.
TransactionMgr_var transactionMgr =
   transactionMgrServant->_this();

// Get reference to the initial naming context
CORBA::Object_var obj =
   orb->resolve_initial_references("NameService");

// Cast the reference from CORBA::Object
```

```
CosNaming::NamingContext_var name_service =
   CosNaming::NamingContext::_narrow(obj);

// Create the name with which the transaction manager will be bound
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Transactions");

// Register transactionMgr object reference in the
// Naming Service at the root context
name_service->bind(name, transactionMgr);
```

一旦事务管理器被注册到Naming Service，客户就可以从Naming Service获得它的对象引用。下面的C++代码展示了客户如何做到这点。

```
// First initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get reference to the initial naming context
CORBA::Object_var obj =
   orb->resolve_initial_references("NameService");

// Cast the reference from CORBA::Object
CosNaming::NamingContext_var name_service =
   CosNaming::NamingContext::_narrow(obj);

// Create the name with which the transactionMgr
// is bound in the Naming Service
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Transactions");

// Resolve transactionMgr from the Naming Service
CORBA::Object_var obj = name_service->resolve(name);

// Cast the reference from CORBA::Object
TransactionMgr_var transactionMgr =
   TransactionMgr::_narrow(obj);
```

一旦到事务管理器的引用被客户获得，客户就可以使用它来调用操作，并且获得到其他CORBA对象和服务的引用了。

变体（Variants）

Self-registering Resources模式。资源提供者和资源等参与者可以以相同的软件工件来实现。在这种情况下，资源和资源提供者之间没有区分，所以注册到查找服务的引用将会是资源引用。例如，分布式服务的资源可以直接把自身注册到查找服务，提供到自身的引用。

Multi-faceted Resource Provider模式。当资源提供者提供了所需获得的多于一个的资源时，被注册的资源提供者的引用也可标识资源。资源提供者为每个通过查找服务发布的资源注册了一个唯一的到其自身的引用。注册到查找服务的引用应当对应于资源提供者，但会间接地引用到该资源提供者提供的一个唯一的资源。

Resource Registrar模式。可以有一个单独的实体负责把引用注册到查找服务，而不必由实际提供资源的实体来做这件事。也就是说，资源提供者不需要是向查找服务注册引用的实体，该

责任可以由另一个称做资源注册者的单独实体来完成。

Federated Lookup模式。可以用查找服务的多个实例来共同建立查找服务的联合体。在联合体中，查找服务的实例协作以向资源使用者提供更广泛的资源。Half-Object Plus Protocol [Mesz95]模式描述了如何在分离查找服务实例的同时依然保持它们的同步性。

可以把联合查找服务配置为当自身无法执行请求时将请求转发给其他服务。这就扩展了查询的范围，并使得资源使用者可以获得对以前无法获得的资源的访问。在联合体中的查找服务可以位于相同或者不同的区域。

Replicated Lookup模式。查找服务可以用来建立容错系统。复制是一个众所周知的概念，可带来容错性。使用查找服务，可以在两个层次上应用复制:

- 首先，查找服务本身可以被复制。查找服务的多个实例可以提供负载平衡和容错处理。Proxy [GoF95]模式可以用来向客户隐藏对查找服务的选择。例如，多个ORB实现在客户端提供了智能代理[HoWo03]，可以用于隐藏在所有查找服务中多个被复制的服务间的选择。
- 其次，资源和资源提供者注册到查找服务的引用也可以复制。查找服务可以扩展成支持资源提供者为相同的属性列表进行多项注册，以CORBA Naming Service为例，可以有相同的名称。查找服务可以用多种策略[GoF95]来配置，以便把资源使用者的请求转发给合适的资源提供者。例如，查找服务可以用轮流的策略来在多个事务管理器的实例中进行选择（这些事务管理器以相同的属性列表注册到查找服务）。Borland使用这种复制[Borl04]来扩展它们的CORBA实现（Visibroker）的可伸缩性。

结果（Consequences）

使用Lookup模式有几个**优点**:

- 可获得性（Availability）。使用查找服务，资源使用者可以根据需要找到在它的环境中哪些资源可用。请注意，资源或者其相应的资源提供者可能会变得不再可获得，但它的引用可能尚未被从查找服务中删去。参见“缺点”中的悬挂引用（Dangling References）以获得更多细节。
- 自举（Bootstrapping）。查找服务使得资源使用者可以获得初始的资源。在分布式系统中，自举协议使得资源使用者可以找到查找服务并用它来找到其他的分布式服务。
- 位置独立性（Location independence）。查找服务提供了位置透明性，这是因为它向资源使用者屏蔽了资源提供者的位置。类似地，该模式也向资源提供者屏蔽了资源使用者的位置。
- 配置简单性（Configuration simplicity）。基于查找服务的分布式系统需要很少或者不需要手工配置。不需要为了分发、发现或者访问远程对象而共享或者传输文件。自举协议的使用是自组网络场景（在这样的场景中环境经常改变并且无从预知）的关键特性。
- 基于属性的选择（Property-based selection）。资源可以基于属性来选择。这样就使得细粒度地匹配用户需求和资源发布成为可能。

使用Lookup模式有几个**缺点**:

- 单一失败点（Single point of failure）。Lookup模式带来的一个后果是，如果查找服务的一个实例崩溃了，那么系统可能会失去注册的引用及其相关属性。等查找服务重启之后，资

源提供者需要重新向其注册资源，除非查找服务具有持久化的状态。这可能既令人厌烦又容易导致错误，因为它要求资源提供者来检测查找服务是否崩溃并且重启。此外，查找服务还可能会成为瓶颈，影响系统的性能。所以更好的解决方案是对查找服务引入复制，这在“变体（Variants）”一节中已经讨论过。

- 悬挂引用（Dangling references）。Look模式带来的另一个后果是产生悬挂引用的危险性。随着相应的资源提供者终止服务或者被移走，注册到查找服务的引用可能会变得过时。在这种情况下，Jini [Sun04c]所用的Leasing模式可以提供帮助，它强迫资源提供者定期地“续借”它们的引用（如果它们不希望自己的引用被自动删除的话）。
- 不想要的复制（Unwanted replication）。当具有相同属性的类似资源进行发布，但又不想进行复制时，就会产生问题。取决于查找服务的实现，相同资源的多个实例可能会错误地注册，或者一个资源提供者可能会覆写前面一个资源提供者的注册。确保至少一个属性是唯一的可以避免这个问题。

已知应用（Known Uses）

CORBA [OMG04a]。Common Object Services Naming Service和Trading Service实现了查找服务。Naming Service的查询语言相当简单，只使用了名字，而Trading Service的查询语言则相当强大，支持对组件的复杂查询。

LDAP [HoSm97]。Lightweight Directory Access Protocol（LDAP）定义了网络协议和信息模型，用于访问信息目录。LDAP服务器允许以文本、二进制数据、公钥证书、URL和引用的形式存储几乎任何类型的信息。LDAP服务器经常受访问权限保护，因为它们包含了关键的信息，因而必须阻止未经授权的访问。LDAP客户可以查询保存在LDAP服务器上的信息。大型组织通常会用LDAP目录来把它们的用户的e-mail、Web和文件共享服务器的数据库集中化。

JNDI [Sun04f]。Java Naming and Directory Interface（JNDI）是Java中的一个接口，为应用程序提供了名字和目录功能。通过使用JNDI，Java应用程序可以存储和获取任意类型的命名的Java对象。此外，JNDI还提供了查询功能，允许资源使用者通过属性来查询Java对象。使用JNDI还可以同已有的名字和目录服务（比如CORBA Naming Service、RMI registry [Sun04g]和LDAP）集成。

Jini [Sun04c]。Jini支持自组网络计算（ad hoc networking），这是通过使得服务可以不需要任何预先计划、安装和人的干预就可以加入网络来做到的。Jini服务注册到Jini查找服务，这些服务会被用户通过Jini发现协议来访问。为了增强网络可靠性，Jini查找服务定期地向潜在客户广播其有效性。

COM+ [Ewal01]。Windows 注册表是一个查找服务，允许资源使用者基于键来获取注册的组件。键既可以是ProgId、GUID（Global Unique IDentifier），也可以是组件的名称和版本。然后注册表允许获得相关联的组件。

UDDI。Universal Description, Discovery, and Integration协议（UDDI）[UDDI04]是Web Service [W3C04]的关键组件之一。UDDI使得Web Service的发布者可以公布他们的服务，而客户则可以搜寻相匹配的Web Service。发布包含了服务描述，并指向定义了服务接口的详细的技术规约。

对等网络。对等（P2P）网络技术，比如JXTA [JXTA04]支持发布和发现对等网络使用者和资源。在P2P环境中的资源经常是文件和服务。

DNS [Tane02]。Domain Name Service（DNS）负责域名和IP地址之间的映射和协调。它由一系列提供映射的名字服务器的层次结构组成。因此，它是联合查找如何工作的好例子。客户查询任意一个临近的名字服务器，发送包含名字的UDP包。如果临近的名字服务器可以解析，那么就返回IP地址，否则的话它就使用精确定义的协议向层次结构中的下一个名字服务器查询。

网格计算[BBL02] [Grid04] [JAP02]。网格计算是关于分布式资源（比如处理时间、存储空间和信息）的共享和组合的。网格包含了多台互联的计算机，它们构成了一个虚拟系统。网格计算使用Lookup模式来寻找分布式资源。取决于项目，系统中查找服务的角色常常被称做“资源中介者”、“资源管理者”或者“发现服务”。

Eclipse插件注册表[IBM04b]。Eclipse是一个开放的、可扩展的IDE，它提供了一个通用的工具平台。它的可扩展性是基于它的插件构架。Eclipse包含一个插件注册表，实现了Lookup模式。插件注册表保存了所有已发现的插件、扩展点以及扩展的列表，允许客户通过标识来定位这些内容。插件注册表本身可以通过几个框架类被客户发现。

电话目录服务。Lookup模式在现实世界中有一个已知应用，就是电话目录服务。某个人X可能想要获取另一个人Y的电话号码。假定Y把他/她的电话号码注册到了查找服务，在这种情况下是电话目录服务。X会拨打目录服务的号码并获得Y的电话号码。电话目录服务有一个众所周知的电话号码，比如411[⊖]，或者是一个网站[ATT04]，这样X就可以联系到它。

接待员[TwAl83]。在一个公司中，接待员可以看做是Lookup模式的真实例子。接待员管理公司中所有职员的联系信息（电话号码的形式）。当有人想要联系某个职员，那么他们会先联系接待员，然后接待员会提供要找的那个人的“引用”。类似地，如果新人加入了公司，或者老人离开了公司，那么通常接待员会更新联系信息以处理未来的查询。

又见（See Also）

Activator模式[Stal00]把激活的组件注册到查找服务，以便向资源使用者提供到它们的访问。在很多情况下，从查找服务获得的引用其实是到工厂的引用，实现Abstract Factory模式[GoF95]。这把组件的位置同它们的激活解耦合了。

Sponsor-Selector模式[Wall97]把选择资源同推荐资源的职责解耦合了，并把这两个职责分别交给了两个参与者：selector和sponsor。Sponsor-Selector模式可以用于Lookup模式以改进对匹配用户需求的资源的查找。sponsor的角色同资源提供者相符，而查找服务则是selector。

Service Locator模式[ACM01]则封装了对EJB [Sun04b] Home对象的JNDI查找以及创建业务对象的复杂性。

致谢（Credits）

我们要向EuroPLoP 2000上我们的把关者Bob Hanmer致谢，谢谢你的反馈和宝贵的评论。我

⊖ 在中国是114。——译者注

们还要感谢所有参加笔者2000年在离开西门子期间在奥地利的St. Martin举办的研讨会的人：Frank Buschmann、Karl Pröse、Douglas C. Schmidt、Dietmar Schütz和Christa Schwanninger，以及参加笔者在EuroPLoP 2000上的研讨会的人： Alexandre Duret-Lutz、Peter Gassmann、Thierry Geraud、Matthias Jung、Pavel Hruby、Nuno Meira、James Noble和Charles Weir，谢谢你们宝贵的评论和建议。

2.2 Lazy Acquisition模式

Lazy Acquisition（延迟获取）模式把资源获取推迟到了系统执行的可能的最后一刻，以便优化资源使用。

实例（Example）

考虑一个医疗影像归档和通信系统（Picture Archiving and Communication System，PACS），它提供了病人数据的存储。数据包含病人的细节情况，比如地址信息以及医疗历史。此外，数据还可以包含数字化的影像，来源可能是X光片、CT。除了需要支持不同的病人数据来源，PACS系统还必须提供对数据的高效访问。这样的数据通常由医师和放射学家为了诊断和治疗目的而访问（见图2-7）。

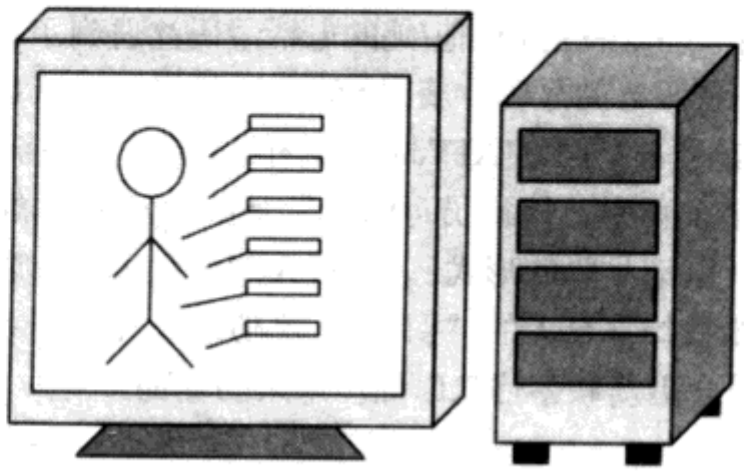


图 2-7

PACS系统通常会以3层构架来建立。中间层维护业务对象，用于表示病人数据。因为数据必须持久化，所以这些业务对象以及它们包含的对象被映射到某种持久化存储，通常是数据库。当医师查询特定的病人时，会取数据并创建相应的业务对象。这个业务对象被传送到系统的表示层，表示层会取出相应的信息并向医师展示。用来获取相关病人数据并建立业务对象所需的时间同病人数据的尺寸成比例。对于一个具有很久的医疗历史以及对应于不同的体检的很多数字化影像的病人，获取所有数据并创建相应的业务对象可能会花去很长时间。因为高性能是这类系统的典型非功能性需求，所以获取病人记录的延迟是个大问题。

PACS系统应该如何设计才能不管病人记录中有多少影像都可以快速获取病人信息呢？

环境（Context）

具有有限资源且必须满足高要求（比如高吞吐和高可用性）的系统。

问题（Problem）

有限的资源可用性是所有的软件系统都面临的约束。此外，如果可用的资源没有正确管理，可能会给系统带来瓶颈，并且对系统的性能和稳定性带来显著影响。为了确保资源在需要时是可用的，大多数系统会在启动时获取资源。但是，早期获取资源会带来很大的获取开销，而且会导致资源浪费（特别是如果资源不是立刻需要的话）。

必须获取和管理昂贵的资源的系统需要一种方式来降低获取资源的初始开销。如果这些系统一开始就获取全部资源，那么会导致很大的开销，会有很多资源不必要地被消费。

为了解决这些问题，需要解决这些作用力：

- 可用性（Availability）。资源的获取应当被控制为尽量减少资源短缺的可能性，并确保当需要时有足够数目的资源可用。
- 稳定性（Stability）。资源短缺会导致系统不稳定，所以资源应当以对系统稳定性具有最小影响的方式被获取。
- 系统快速启动（Quick system start-up）。在系统启动时对资源的获取应当被以优化系统启动时间的方式实现。
- 透明性（Transparency）。解决方案对资源使用者应当是透明的。

解决方案（Solution）

在可能的最后一刻获取资源。除非不可避免，否则不去获取资源。当资源使用者一开始要求获取资源时，创建并返回一个资源代理。当资源使用者试图访问资源时，资源代理获取实际资源并把资源使用者的访问请求重定向到资源。因此，资源使用者依赖于资源代理，但既然资源代理提供的接口同资源一样，那么对于资源使用者而言，访问的是资源还是资源代理是透明的。

通过用代理来表示可能获取起来代价昂贵的资源，获取一组资源的总体代价可以最小化，而且，因为不在一开始就获取大量资源，所以需要同时管理的资源总量也最小化了。

结构（Structure）

下列参与者构成了Lazy Acquisition模式的结构：

- 资源使用者获取并使用资源。
- 资源是一个连接或内存之类的实体。
- 资源代理（Resource Proxy）截获资源使用者对资源的获取，并把延迟获取的资源交给资源使用者。
- 资源提供者管理并提供多种资源。

下面的CRC卡片描述了参与者的职责与协作（见图2-8）。

下面的类图描述了Lazy Acquisition模式的结构（见图2-9）。

类图表明，资源使用者依赖于资源代理。因为资源代理提供了和资源相同的接口，所以访问的是资源代理还是资源对于使用者而言是透明的。

动态（Dynamics）

场景1：在这个交互场景中，资源提供者不仅提供资源，还扮演了创建资源代理的工厂的角

Class Resource User	Collaborator • Resource Proxy
Responsibility • 获取并使用资源	

Class Resource	Collaborator
Responsibility • 通过资源代理被获取并使用	

Class Resource Proxy	Collaborator • Resource • Resource Provider
Responsibility • 假装是资源 • 提供和资源一样的接口 • 使得资源提供者的实际资源可用	

Class Resource Provider	Collaborator • Resource
Responsibility • 管理资源并向资源代理提供资源	

图 2-8

色（见图2-10）。

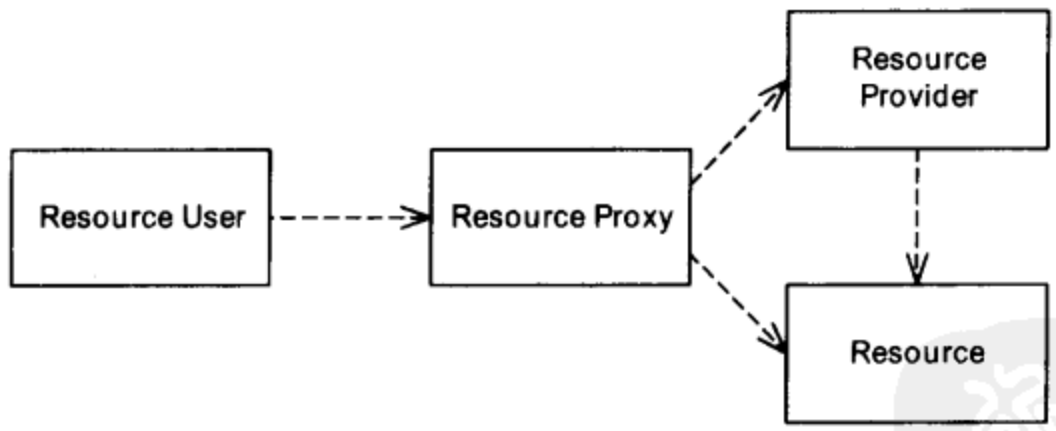


图 2-9

当资源使用者试图从资源提供者获取资源时，资源提供者创建并返回一个资源代理给使用者，而不是返回实际资源。

场景2：Lazy Acquisition模式的关键是资源代理在第一次被资源使用者访问的时候。一开始，资源代理不拥有资源。第一次访问时才会获取资源。所有后续的对资源的访问都被资源代理传递给实际资源。资源使用者不会注意到资源代理提供的间接层次（见图2-11）。

实现（Implementation）

这个模式的实现可以用下列步骤来描述：

1）确认需要被延迟获取的资源。通过性能分析和系统分析，确认具有下列一个或多个属性

的资源:

- 获取起来代价昂贵的资源。
- 数量有限的资源。
- 在获取后很久未用的资源。

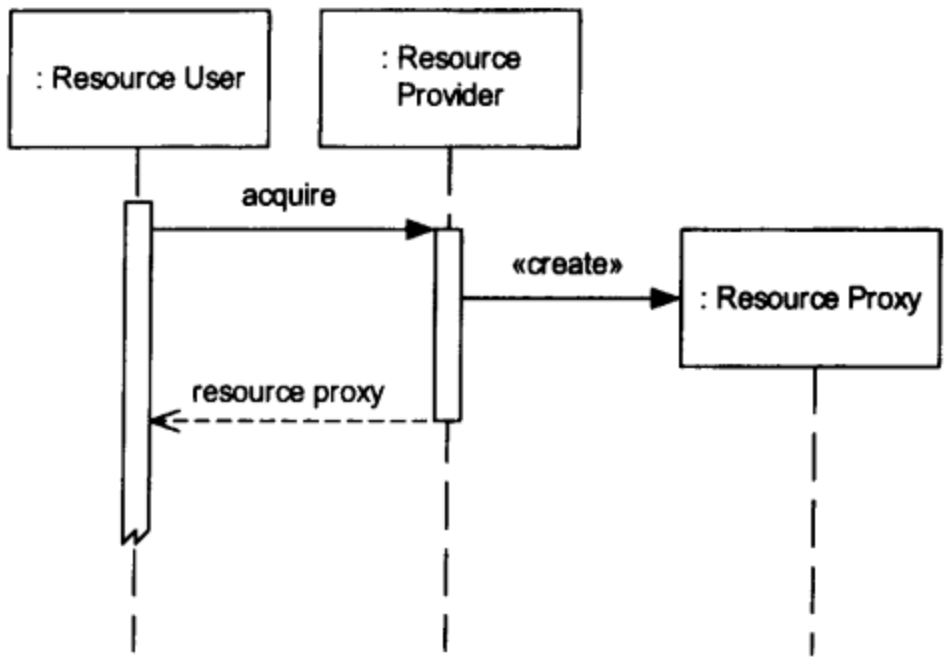


图 2-10

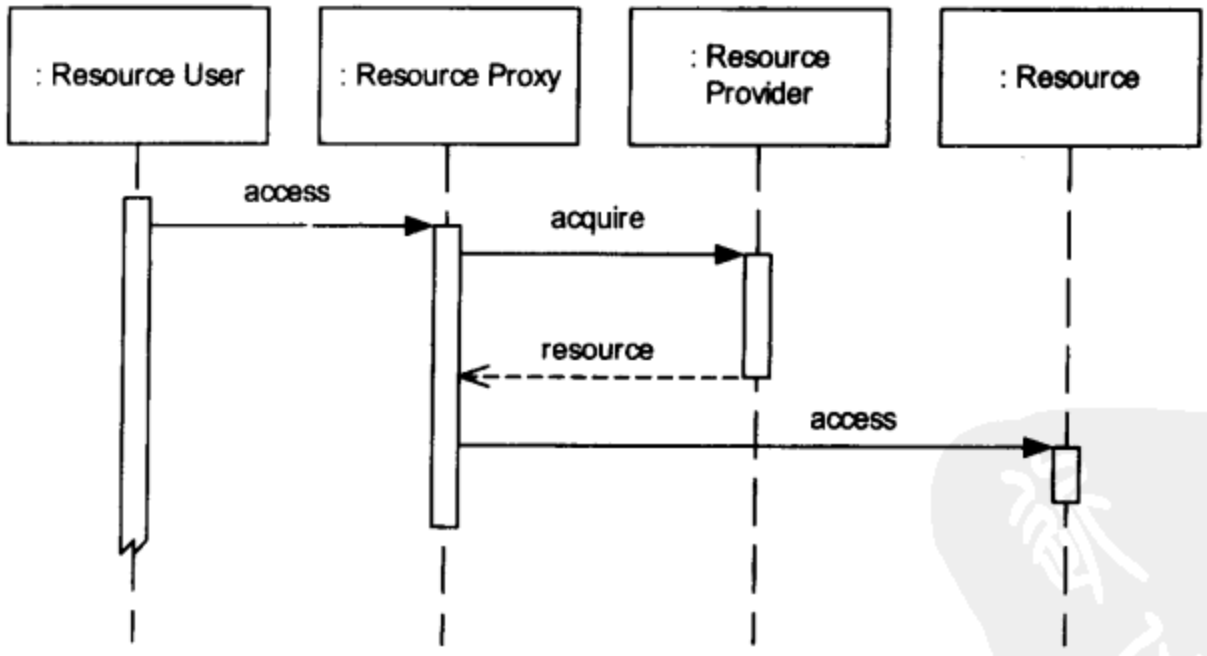


图 2-11

审视找出的每个资源及其使用情况，并决定如果改用Lazy Acquisition模式，总体资源可用性、系统稳定性和系统启动时间能否改善。对每个找出的资源应用后面的实现步骤。

2) 定义资源代理接口。对于需要延迟获取的每个资源，都需要定义一个资源代理为Virtual Proxy [GoF95] [POSA1]，它的接口同资源的接口完全一样。

```
interface Resource {
   public void method1 ();
   public void method2 ();
```

```
}

public class ResourceProxy implements Resource {
   public void method1 () {
       // ...
   }
   public void method2 () {
       // ...
   }
}
```

3) 实现资源代理。实现资源代理，使之隐藏对资源的延迟获取。资源代理只在资源使用者实际访问资源时才获取资源。一旦实际资源被获取，资源代理就使用委托来处理所有的资源请求。资源代理也可以负责在获取资源后将其初始化。

```
public class ResourceProxy implements Resource {
   public void method1 () {
       if (!acquired)
           acquireResource ();
       resource.method1 ();
   }
   public void method2 () {
       if (!acquired)
           acquireResource ();
       resource.method2 ();
   }
   private void acquireResource () {
       // ...
       acquired = true;
   }
   private boolean acquired = false;
}
```

4) 定义获取策略。定义资源代理从资源提供者实际获取资源的策略。可以用Strategy模式[GoF95]来配置不同类型的策略。简单的策略可以是把资源的获取延迟至资源使用者访问资源。其他的策略可以是基于某个状态机来获取资源。例如，某个组件的实例化可能触发另一个已经存在的组件的资源代理去获得相应的资源。此外，资源代理还可以提供关闭延迟获取机制的能力，这样资源就会被立即获取。

在代码示例中资源使用者可以用acquireResource()方法来显式地让资源代理去获取资源。

```
public class ResourceProxy implements Resource {
   // ...
   void acquireResource () {
       resource = new ResourceX ();
   }
   ResourceX resource;
}
```

在资源代理中配置资源获取策略以获取资源。

5) 为延迟获取的资源实现正确的资源释放机制。如果资源用户没有显式释放资源，那么或者使用Evictor模式或者使用Leasing模式来自动释放资源。这样一来，被延迟获取的资源分别需

要实现EvictionInterface或者注册到租赁提供者。

实例解析（Example Resolved）

考虑一下医疗影像归档和通信系统（PACS）的例子。为了解决快速获取病人信息的问题，应该使用Lazy Acquisition模式。

当发起一个请求以获取关于一个特定病人的全部信息时，建立的查询不会获取任何影像数据。在返回的业务对象中，对病人记录中的每幅影像都会创建一个代理。表示层会处理业务对象并展示所有信息。对于遇到的所有影像代理，它在建立的表示中都会创建链接。当需要查看这样的一个链接所对应的影像时，才会（延迟地）从文件系统中去获取。影像直接保存于文件系统，因为把影像的大量二进制数据保存在数据库中通常是低效的。

使用这个解决方案可以优化对病人的文字数据的获取。通常文字数据量不会很大。这可以为医师提供有关病人医疗记录的一个很好的摘要。如果医师希望看到病人记录中的影像，那么可以按需要去延时获取。

下面的示例代码展示了PatientManager类，这个类会查询数据库以获取病人数据。它获取除影像之外的全部数据。

```
public class PatientManager {
   public static PatientRecord getPatientRecord (String patientId) {
       return dbWrapper.getRecordWithoutImages (patientId);
   }
}
```

返回的PatientRecord保存有一个MedicalExams列表，这个列表中的每一项都会引用一幅影像。MedicalExam类无法区分正常装载的影像和延迟装载的影像，因为它们的接口是一样的。

```
interface PatientRecord
{
   String getName ();
   String getAddress ();

   List getMedicalExams ();
}

interface MedicalExam
{
   Date getDate ();
   // Get the digitized image for this exam
   Image getImage ();
}

interface Image
{
  byte [] getData ();
}
```

可以用多种策略来实现Image接口。下面的ImageProxy类只在实际访问时才从文件系统装载影像，从而实现了对影像的延迟获取。

```
public class ImageProxy implements Image
{
   ImageProxy (FileSystem aFileSystem, int anImageId) {
       fileSystem = aFileSystem;
       imageId = anImageId;
   }

   public byte [] getData () {
       if (data == null) {
           // Fetch the image lazily using the stored ID
           data = fileSystem.getImage (imageId);
       }
       return data;
   }

   byte data[];
   FileSystem fileSystem;
   int imageId;
}
```

请注意，ImageProxy需要拥有如何从文件系统中得到延迟获取的影像的信息。

特化（Specializations）

一些从Lazy Acquisition模式派生而来的特化模式是：

Lazy Instantiation模式[BiWa04b]。延迟对象/组件的实例化，直到用户访问实例。因为对象实例化经常会牵涉到动态内存分配，而内存分配通常是很昂贵的，所以延迟实例化可以节省初期开销（特别是对于未被访问的对象）但是，在大批用户同步访问对象，导致高需求的情形时，使用延迟实例化可能会带来很大的开销。

Lazy Loading模式。延迟加载一个共享库，直至要访问包含于这个共享库的程序元素。可以使用Component Configurator模式[POSA2]来实现。Lazy Loading模式常常和Lazy Instantiation模式结合使用，因为当对象被加载时需要实例化。Lazy Load模式[Fowl02]描述了如何把从数据库中加载对象状态推迟到客户对对象实际产生“兴趣”的时候。

Lazy State模式[MoOh97]。延迟初始化对象的状态，直至这个状态被访问到。Lazy State模式经常用于有大量状态信息但是访问频率很低的情形。这个模式同Flyweight模式[GoF95]或者Memento模式[GoF95]结合时会更有用。在对象的网络中，Lazy Propagator模式[FeTi97]描述了相互依赖的对象如何判断它们是否受状态变化的影响并需要更新自己的状态。

Lazy Evaluation模式[Pryc02]。延迟求值意味着一个表达式的值只有在计算另一个表达式时需要用到的时候才会被计算出来。对参数延迟求值使得函数可以被部分求值，这样就可以把简化过的函数用于剩余的参数。使用延迟求值可以显著改进求值的性能。因为不需要的计算被避免了。在Java或者C++这样的编程语言中，布尔表达式的子条件的求值以短路操作符（比如&&）的形式运用了延迟求值。

Lazy Initialization模式[Beck97]。当你的程序的某个部分被第一次访问时才会初始化。这个模式具有在某些情形下避免开销的优点，但是缺点是增加了访问程序的未被初始化部分的机会。

Variable Allocation模式[NoWe00]。只有在需要时才分配和释放尺寸可变的对象。这一特化是把Lazy Acquisition模式运用于内存的分配和释放。

变体（Variants）

Semi-Lazy Acquisition模式。除了尽可能早或者尽可能晚地获取资源，资源还可以在其他3种时刻被获取。其背后的想法是，不要在一开始就获取资源，但也不要等到实际需要资源的时候才获取资源。可以在两个极端之间的某个时刻获取资源。一个例子是网络管理系统需要建立网络拓扑树的时候。

有3种可能的选择：

- 在应用程序启动的时候创建。
 - 赞同的理由：当应用程序初始化完毕后树即可用。
 - 反对的理由：漫长的启动时间。
- 在用户请求时创建。
 - 赞同的理由：启动时间很短。
 - 反对的理由：用户需要等待树被创建。
- 在应用程序启动之后、用户请求之前创建。
 - 赞同的理由：启动时间很短，需要时树已建好。

在网络管理系统中，经常使用最后一种选择。

结果（Consequences）

使用Lazy acquisition模式有几个**优点**：

- 可用性（Availability）。使用Lazy Acquisition模式确保了所有资源都不会在一开始就获取。这有助于把系统缺乏资源以及获取不需要的资源的可能性降至最低。
- 稳定性（Stability）。使用Lazy Acquisition模式确保了资源只在需要时才会被获取。这避免了在一开始不必要地获取资源，从而降低了资源耗尽的可能性，使得系统更稳定。
- 优化系统的启动时间（Optimal system start-up）。使用Lazy Acquisition模式确保了不是立即需要的资源会在较晚的时候才被获取，这有助于优化系统的启动时间。
- 透明性（Transparency）。使用Lazy Acquisition模式对资源使用者而言是透明的。资源代理向资源使用者隐藏了实际的资源获取操作。

使用Lazy acquisition模式有几个**缺点**：

- 空间开销（Space overhead）。这个模式会带来少许空间开销，因为间接性所需的代理需要额外的内存。
- 时间开销（Time overhead）。执行延迟获取会在获取资源时造成显著的延时，并且由于额外的间接层次的缘故，也会对正常的程序执行带来时间开销。对实时系统而言，这样的行为可能是无法接受的。
- 可预测性（Predictability）。延迟获取的系统的行为可能会变得无法预期。如果系统的多个部分都尽可能晚地获取资源，那么当系统的所有部分都同时尝试获取资源时性能曲线会有陡峭的变动。

已知应用（Known Uses）

Singleton。Singleton [GoF95]是指系统中唯一的对象，通常会延迟实例化。在某些情况下，

Singleton会被几个线程访问。Double-Checked Locking惯用法[POSA2]可以用于避免在实例化时出现几个线程之间竞争的情况。

Haskell [Thom99]。Haskell语言和其他函数式编程语言一样，也允许对表达式延迟求值。Haskell只对为获得结果而不得不计算的程序部分求值。使用需求驱动的求值，Haskell中的数据结构只进行足以得出答案的计算，其中有一部分可能根本不会被计算到。

Java 2 platform, Enterprise Edition (J2EE) [Sun04b]。在J2EE应用程序服务器[Iona04]中，Enterprise JavaBeans（EJB）容器会同时容纳很多不同的组件。为了避免资源耗竭，它们需要确保只有实际被客户使用的组件才处于激活状态，其他组件则不应激活。典型的解决方案是对组件及其属性使用Lazy Loading和Lazy Instantiation模式。这节约了宝贵的资源，并保证了可伸缩性。同样，很多J2EE应用服务器也只在Java Server Pages（JSP）被实际访问时（而不是被部署时）才会把它们编译成servlet。

自组网络计算[Sun04c] [UPnP04]。在自组网络环境中，设备和它们的组件之间只存在临时性的关系，所以持有当时并不实际需要的资源的代价过于高昂。这意味着组件需要不断地装载、实例化、销毁、卸载。所以，自组网络计算框架需要提供延迟装载和延迟实例化的机制。执行对设备的延迟发现也是有可能的，只有下层框架前一次执行设备探索后设备列表有了改变，应用程序才会得到通知[IrDA04]。

操作系统。直到需要时才完整装载应用程序，这是操作系统的常见行为。例如，在大多数Unix系统中，比如Solaris [Sun04h]和Linux，可以用一个环境变量LD_BIND_NOW来说明共享对象是否应当用延迟模型来加载。在延迟加载模型中，任何被标记为延迟加载的外部依赖都只有被显式引用到时才会加载。通过利用函数调用的延迟加载，外部依赖的加载会被延迟到第一次引用的时候。这样有个额外的好处是，从未被引用的对象就不会被加载。

.NET Remoting [Ramm02]。在.NET Remoting中，所谓的“单件远程对象”（singleton remote object）是指可以被多个客户使用但在服务器上一个对象类型同时只能存在一个实例的对象。这些单件远程对象只在第一次访问时才会实例化（即便客户可能在第一次访问它们之前就获取了引用）。

COM+ [Ewal01]。即时（Just-in-Time，JIT）激活是COM+提供的一项自动服务，有助于更有效地使用服务器资源，特别是当把你的应用程序扩展为处理大量事务的时候更是如此。当组件被配置成“JIT激活”的时候，COM+会不时地停止一个实例，而客户依然持有该对象的激活引用。下次客户调用该对象的方法时（客户依然以为对象是激活的），COM+会透明地重新为客户即时激活该对象。基于COM+和.NET的应用程序支持JIT激活。.NET应用程序必须设置System.EnterpriseServices包的配置属性。

JIT编译。今天的Java虚拟机（JVM）广泛使用了JIT编译。把通常形式的Java字节码编译成快速的特定机器的汇编代码是即时完成的。支持这一特性的虚拟机之一是IBM J9 JVM [IBM02]。同JIT编译相对的是预先（ahead-of time，AOT）编译。

Java [Sun04a]。JVM实现通常通过在第一次运行一个类的代码时才加载这个类来优化Java类的加载。这一行为很明显遵循Lazy Loading模式。

制造[VBW97]。很多行业（比如汽车工业）已经采用的“即时制造”也遵循同样的模式。

装配的部件只在需要时才制造。这节约了固定仓储的开销。

Eclipse插件[IBM04b]。Eclipse是一个通用工具平台，一个海纳百川的、开放的可扩展IDE。它的扩展性基于一个允许任意用户成为插件贡献者的插件构架。插件声明决定了插件的可视化特性是一开始就加载的，而包含在JAR（Java archive）中的实际逻辑则是延迟加载的，只有在第一次使用插件的功能时才会加载。

堆压缩。[GKVI+03]的研究涉及Java语言的一个特殊的垃圾收集器，使得使用小于应用程序实际尺寸的堆成为可能，这是通过临时性地压缩堆中不用的对象来做到的。当再次访问这样的对象时，它会延迟地解压对象的一部分，分配需要的内存。

FrameMaker [Adob04]。桌面出版程序FrameMaker只有在第一次显示包含相应图片的页面时才会打开并读取该图片文件。然后，图形被绘制到屏幕上，被绘制的图形被保存在临时存储空间中。当页面再次显示时，FrameMaker会检查被引用的文件是否更新过。如果没有，那么就会重用临时存储空间中已经绘制好的图形。

又见（See Also）

可以认为，Eager Acquisition模式同Lazy Acquisition模式正相反。Eager Acquisition描述的概念是一开始就获取资源以避免客户在第一次访问资源时的获取开销。

因为在某些用例中Lazy Acquisition模式和Eager Acquisition模式都不是最优的，所以Pooling模式把两者结合成一个模式以优化资源使用。

Lazy Optimization模式[Auer96]有助于在程序已经正常运行并且系统设计反映了对程序应当如何组织的最佳理解的时候优化性能。

Thread-Specific Storage模式[POSA2]使用代理来屏蔽创建用于在每个线程的对象集合中唯一地标识相关联的线程相关对象的key。如果代理在访问时还没有key，那么它会让key factory创建一个新key。

致谢（Credits）

感谢Frank Buschmann和Markus Völter对这个模式的早期版本的宝贵意见。特别要感谢EuroPLoP 2002上我们的把关者Kevlin Henney，以及参与笔者的研讨会的人们：Eduardo Fernandez、Titos Saridakis、Peter Sommerlad和Egon Wuchner。

我们的技术编辑Steve Rickaby向我们介绍了如何用FrameMaker。整本书的撰写和制作都是用FrameMaker完成的。

2.3 Eager Acquisition模式

Eager Acquisition（预先获取）模式描述了如何通过在实际使用资源前预先获取并初始化资源来使运行时资源获取具有可预测性和快速性。

实例（Example）

考虑一下具有软实时约束（比如可预测性和低的操作延时）的嵌入式电信应用程序（见图2-12）。假定，主要出于价格和可移植性方面的考虑，应用程序被部署到一个商业化的现成的操作系统上

（比如Linux）。

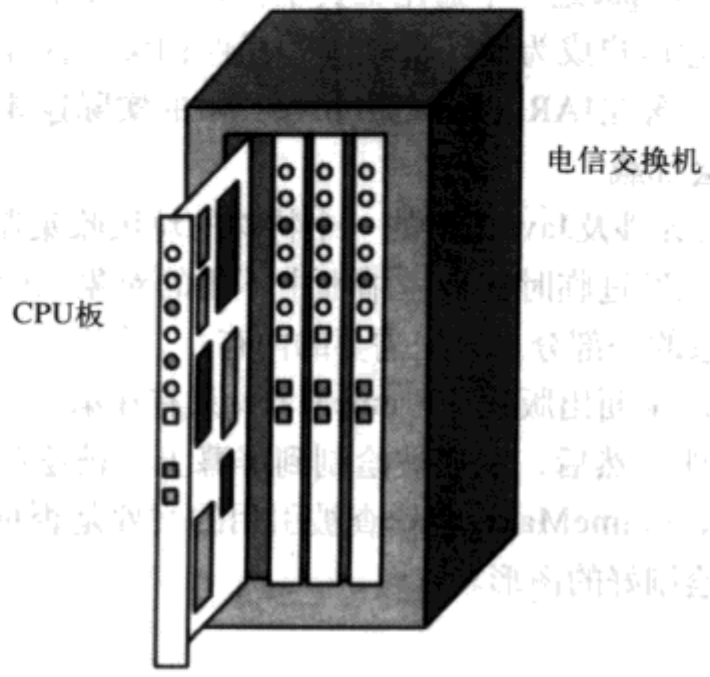


图 2-12

在大多数操作系统中，动态分配内存之类的操作代价都是很高昂的。使用new()或者malloc()之类的内存分配操作所花的时间取决于操作的实现。在大多数操作系统（包括实时操作系统）中，执行动态内存分配的时间是不确定的。

主要理由如下:

- 内存分配受同步原语保护。
- 内存管理（比如对小块内存碎片的整理）会占用时间。

如果不经常进行内存整理，那么内存分配很容易造成内存碎片化。但是，大多数操作系统（包括VxWorks [Wind04]之类的一些RTOS），都不提供这样的内存管理，于是应用程序就可能会受内存碎片化之苦。

背景（Context）

系统获取资源时必须具有高可预测性以及高性能。

问题（Problem）

具有软实时约束的系统需要严格规定何时以及如何获取资源。这类系统的例子有：关键工业系统、高可伸缩性的Web应用程序，甚至桌面应用程序的图形用户界面（GUI）。在每个例子中，这样的系统的用户都会对系统的可预测性、延迟和性能作出一些假定。例如，对桌面应用程序的GUI，用户会期待快速的响应，任何响应延迟都可能会让用户生气。但是，如果执行任何用户发出的请求都会导致昂贵的资源获取（比如动态获取线程和内存），那么就可能会带来不可预期的时间开销。具有软实时约束的系统如何获取资源且依然可以满足约束呢？

为了解决问题，需要解决下面的作用力:

- 性能（Performance）。资源使用者获取资源时必须要快。
- 可预测性（Predictability）。资源使用者获取资源必须具有可预测性——每次获取资源时花

的时间应该一样。

- 初始化开销（Initialization overhead）。需要避免在应用程序运行时执行资源初始化。
- 稳定性（Stability）。需要避免在应用程序运行时资源耗尽。
- 公平性（Fairness）。解决方案必须对其他试图获取资源的资源使用者公平。

解决方案（Solution）

在实际使用之前先获取资源。在使用资源之前（最好是在启动时），资源提供者代理已经从资源提供者那里预先获取了资源。然后，资源就被保存在高效的容器中。当资源提供者代理截获来自资源使用者的获取资源请求时，就可以访问容器并返回请求的资源。

获取资源的时间可以用不同的策略来配置。这些策略应该考虑到不同的因素，比如何时会实际使用资源、资源的数目、资源的依赖性、以及获取资源会花多长时间。可能的选择有在系统启动时获取，或者在系统启动完之后一个特定的可能是计算好的时间来获取。不管使用何种策略，目标都是确保在实际使用资源之前获取资源并且资源可用。

结构（Structure）

下列参与者形成了Eager Acquisition模式的结构:

- 资源使用者获取并使用资源。
- 资源是一个内存或线程之类的实体。
- 提供者代理截获用户对资源的获取请求，并立刻把一开始就获取好的资源交给资源使用者。
- 资源提供者管理并提供多种资源。

下面的CRC卡片描述了参与者的职责与协作（见图2-13）。

Class Resource User	Collaborator • Resource • Provider Proxy
Responsibility • 获得并使用资源	

Class Resource	Collaborator
Responsibility • 被提供者代理从资源提供者获得，并被资源使用者使用	

Class Provider Proxy	Collaborator • Resource • Resource Provider
Responsibility • 提供同资源提供者一样的接口 • 截获资源使用者的资源获取操作，并且以常数时间返回预先获取好的资源	

Class Resource Provider	Collaborator • Resource
Responsibility • 管理资源并向资源使用者提供资源	

图 2-13

下面的类图把Eager Acquisition模式的结构可视化了（见图2-14）。

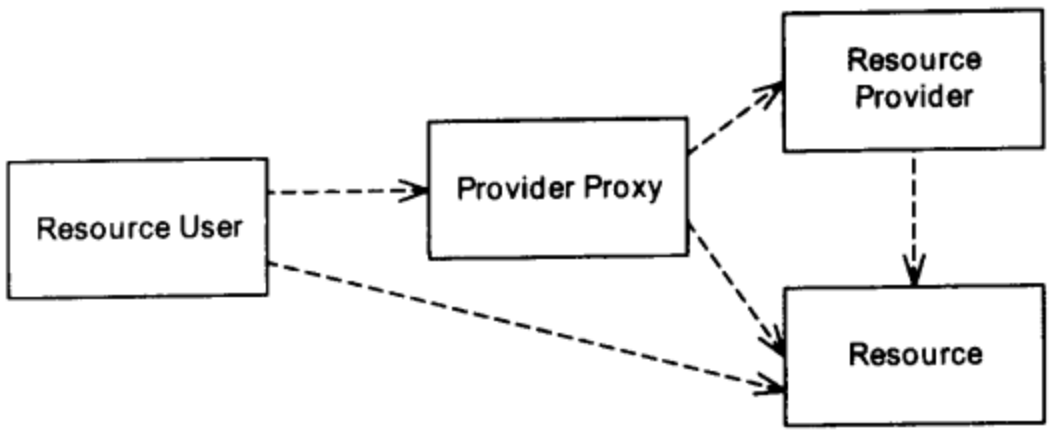


图 2-14

资源使用者并不从资源提供者处直接获取资源，而是通过提供者代理来获取。

动态（Dynamics）

场景1：资源使用者创建提供者代理，然后从这个代理来获取资源。提供者代理在使用资源之前预先获得了资源，最晚会在资源使用者实际试图获取资源时获得（见图2-15）。

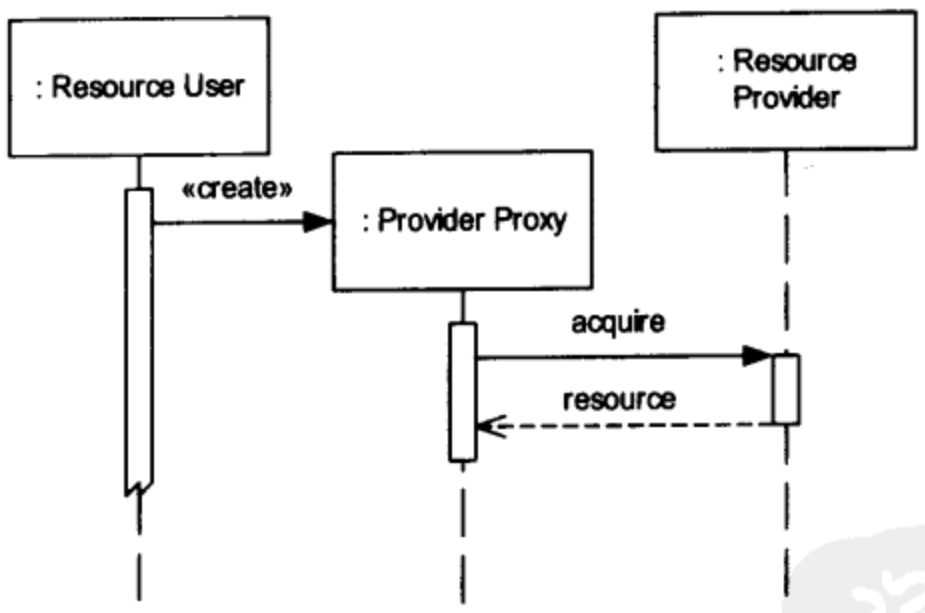


图 2-15

场景2：资源使用者获取资源，但是被提供者代理所截获。下面的顺序图展示了资源使用者如何获取资源（见图2-16）。

提供者代理截获了获取请求，并返回预先获取好的资源。于是资源使用者即可访问并使用资源。

实现（Implementation）

实现Eager Acquisition模式需要六步：

1）选择需要预先获取的资源。决定需要预先获取的资源的类型，以便确保整个系统的行为的可预测性。判定那些昂贵的资源，比如连接、内存和线程。获取这样的资源最可能带来不可

预测性，所以它们是理想的预先获取候选者。

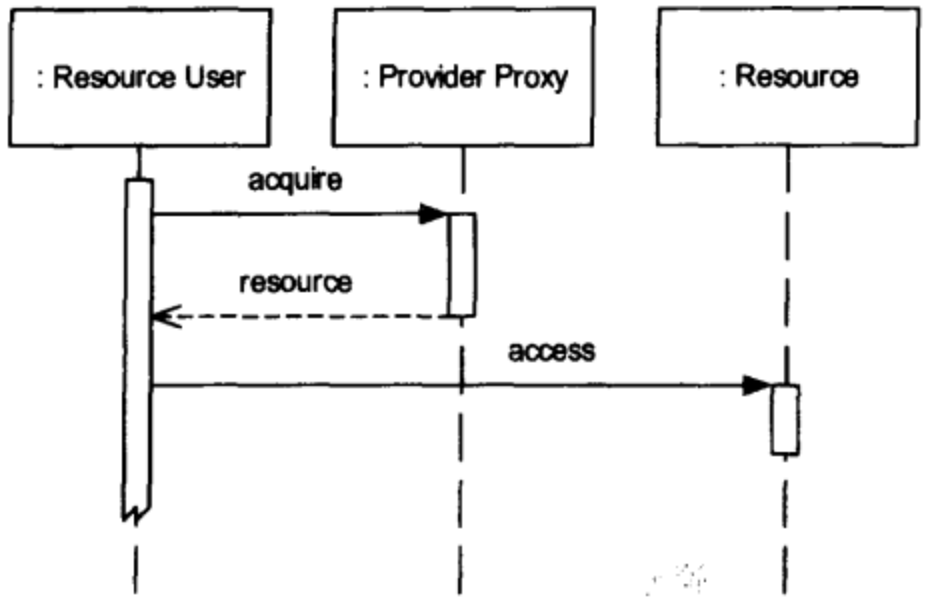


图 2-16

2) 估计资源用量。估计用户在生命周期中会获取多少资源。如果事前并不知道，那么进行试运行以衡量最大的资源用量。如果你无法预测资源用量，那么将其变为一个可配置的参数，从而可以在部署之后简单地调整。为操作者提供关于预先获取的资源的实际使用情况的信息，比如日志之类。

3) 实现提供者代理。提供者代理负责透明地集成Eager Acquisition模式。在实际的设计中，可以有几种方式来包含它；比如使用Virtual Proxy模式[GoF95] [POSA1]或者Interceptor模式[POSA2]。资源使用者可以从Abstract Factory [GoF95]获取提供者代理。如果不需要透明性，那么资源使用者可以直接从资源提供者获取资源，在这种情况下不需要提供者代理。

4) 实现容器。实现一个容器，比如哈希表，用于存放预先获取的资源。容器应当支持可预测的查找，比如哈希表可以提供常数时间的复杂度查找。

5) 决定时间策略。就提供者代理何时预先获取资源选定策略：

- 在系统启动时。实现一个钩子，使得预先获取资源的代码在启动时就被执行。这样做的好处是不会影响系统的运行时行为，但是这要求可以预测资源的使用。
- 在运行时抢先获取。用Reflection模式[POSA1]来检测可能会导致资源使用者在未来需要获取资源的系统状态。抢先行为的好处是可以更贴切地处理获取资源的需求。但是，抢先获取行为的代价是更高的复杂性以及持续监控系统状态的执行开销。

6) 决定初始化语义。决定如何初始化获得的资源，以避免额外的初始化开销。对于某些资源，让资源提供者来在获得时执行完全的初始化是不可能的。在这样的情况下，需要考虑运行时的初始化开销。

实例解析（Example Resolved）

思考一下前面描述过的嵌入式电信应用程序。为了让应用程序更具可预测性，有3种可能的选择：

- 把应用程序对象实现为全局变量，这本质上是预先实例化对象。

- 把对象放在栈上，从根本上避免动态分配内存。
- 实现一个内存池，在系统启动之后但在第一次需要分配动态内存之前预先从操作系统分配好内存。应用程序的对象从内存池获取内存。

第1种选择的缺点是失去了对内存的初始化和释放的控制。而且，很多开发者认为这是糟糕的设计，因为给系统引入了很高的耦合性。类的静态变量使得定义的位置更明显，但是依然具有相同的耦合问题，也具有关于初始化或者释放内存的时间的问题。使用类的静态变量还规定了类的实例数目，这降低了软件的适应性，也降低了软件支持运行时/加载时可变性的能力。

第2种选择，只同分配在栈上的对象打交道，则要求更大的栈尺寸。而且，对象的生命周期必须同调用栈对应，否则的话可能对象的生命周期已经结束但还在被使用。

第3种选择则在应用程序对象初始化问题上灵活得多，但也有一些限制。前面说过，同步开销和对不同尺寸内存块的整理是内存分配时间可预测性不佳的主要原因。内存池只有能够避免对不同尺寸的内存分配的同步和管理才能比操作系统具有更佳的可预测性。

为了避免同步开销，内存池要针对专门的线程，或者用Thread-Specific Storage[POSA2]来内部解决线程相关问题。关于细节，请参见Thread-Local Memory Pool模式[Somm02]。

下面的C++类为固定尺寸的内存块实现了不带同步的内存池。它预期只被用于单线程的环境。它提供的内存由其构造函数预先获得。若要支持多种内存块尺寸，内存池内部对块的管理需要扩展，或者实例化多个独立的内存池，每个内存池提供不同尺寸的块。⊖

```
class Memory_Pool {
public:
Memory_Pool (std::size_t block_size, std::size_t num_blocks)
: memory_block_ (::operator new (block_size * num_blocks)),
    block_size_ (block_size),
    num_blocks_ (num_blocks)
{
    for (std::size_t i = 0; i < num_blocks; i++)
    {
        void *block = static_cast<char*>(memory_block_) +
            i*block_size;
        free_list_.push_back (block);
    }
}

void *acquire (size_t size)
{
    if (size > block_size_ || free_list_.empty())
    {
        // if attempts are made to acquire blocks larger
        // than the supported size, or the pool is exhausted,
        // throw bad_alloc
        throw std::bad_alloc ();
    }
    else
    {
```

⊖ SGI STL的默认allocator（alloc）为小块内存提供了一个线程安全的、支持不同尺寸的内存块的内存池实现，可借鉴。——译者注

```
            void *acquired_block = free_list_.front ();
            free_list_.pop_front ();
            return acquired_block;
        }
    }

    void release (void *block)
    {
        free_list_.push_back (block);
    }


private:
    void *memory_block_;
    std::size_t block_size_;
    std::size_t num_blocks_;
    std::list<void *> free_list_;
};
```

应当为每个需要动态分配内存的线程提供一个Memory_Pool类的实例。构造函数中对内存块的预先获取，以及在acquire()方法中对成块内存的获取可能会抛出bad_alloc异常。

```
const std::size_t block_size = 1024;
const std::size_t num_blocks = 32;
// Assume My_Struct is a complex data structure
struct My_Struct {
    int member;
    // ...
};

int main (int argc, char *argv[])
{
    try
    {
        Memory_Pool memory_pool (block_size, num_blocks);

        // ...

        My_Struct *my_struct =
            (My_Struct *) memory_pool.acquire (sizeof(My_Struct));

        my_struct->member = 42;
        // ...
    }
    catch (std::bad_alloc &)
    {
        std::cerr << "Error in allocating memory" << std::endl;
        return 1;
    }
    return 0;
}
```

acquire()方法使用了Memory_Pool的构造函数中预先获取的内存。理想情况下，构造函数应该一开始就分配足够的内存，这样就可以满足后面acquire()的请求了。但是，如果没有足够的内存可以满足acquire()方法的请求，那么就需要从操作系统获取额外的内存。所以，acquire()还需

要处理这样的情况。当然，只实现获取是不够的，我们还需要正确地实现释放方法 [Henn03] 。

上面的代码只使用于同C语言类似的结构。对于真正的类，内存池必须同new和delete操作符更好地整合，比如这样:

```
void *operator new(std::size_t size, Memory_Pool &pool)
{
   return pool.acquire(size);
}
```

如此一来，上面的结构就可以这样来分配:

```
My_Struct *my_struct_a = new(memory_pool) My_Struct;
```

在这个特定的例子中，优化的代价是new/delete的对称性被打破了。要显式地释放对象并把内存返回给内存池，需要这样做:

```
my_struct->~My_Struct();
memory_pool.release(my_struct);
```

对于如何整合自定义内存管理技术的更详尽讨论，请参见[Meye98]。

虽然上面的实现做了一些假定并带来了一些限制，但是它有优点：提高了动态内存分配的可预测性。此外，通过只分配固定大小的块，还可以避免内存碎片化。关于其他的实现，请参考ACE的内存池实现[Schm02] [Schm03a]以及Boost [Boos04]。

特化（Specializations）

下面是Eager Acquisition模式的一些特化:

Eager Instantiation模式。在这种情况下，对象预先实例化并被容器所管理。当应用程序作为资源使用者请求新对象的时候，可以从列表中传递出新的实例。

Eager Loading模式。Eager Loading模式是将Eager Acquisition模式用于库的加载，比如Unix平台上的共享对象或者Win32上的动态链接库。库在一开始就被加载，这同Lazy Acquisition模式形成了对比。

变体（Variants）

Static Allocation模式。Static Allocation模式也叫做Fixed Allocation模式[NoWe00]，或者叫Pre-Allocation模式，它是把Eager Acquisition模式用于内存的分配。Fixed Allocation模式在嵌入式和实时系统中特别有用。在这样的系统中，内存碎片化和系统行为的可预测性要比动态内存分配更重要。

Proactive Resource Allocation模式[Cros02]。可以基于从资源使用情况推断出的迹象（通过反射技术，而不是仅仅基于估计）来预先获取内存。

结果（Consequences）

使用Eager Acquisition模式有几个**优点**:

- **可预测性**（Predictability）。资源的可获得性是可预测的，因为来自用户的获取资源请求会被截获并即时处理。这就避免了操作系统造成的获取资源时不确定的延时之类的问题。
- **性能**（Performance）。因为资源在需要时已经可用，所以可以很快地用确定的时间来获取。

- 灵活性（Flexibility）。可以很容易地定制资源获取策略。截获用户的资源获取请求可以让提供者代理策略化地获取资源。这对于避免内存碎片化之类的副作用很有帮助。
- 透明性（Transparency）。因为资源是从资源提供者预先获取的，不需要使用者的干预，所以这一解决方案对使用者是透明的。

使用Eager Acquisition模式有几个**缺点**:

- 管理职责（Management responsibility）。管理预先获取的资源成了一个重要方面，因为不是所有的资源都会立刻同资源使用者相关联，所以就需要管理。可以用Caching模式和Pooling模式来提供可能的管理方案。
- 静态配置（Static configuration）。系统变得更为静态化，因为必须预先估计需要的资源数目。必须避免过分预取资源，以保证资源的公平利用并避免资源枯竭。
- 过度获取（Over-acquisition）。子系统可能会预先获取太多的资源，而实际上用不了那么多。这可能会导致不必要的资源枯竭。但是，正确调整好的资源获取策略有助于解决这个问题。也可以用Pooling模式来限制预先获取的资源。
- 减慢系统启动（Slow system start-up）。如果系统启动时需要获取很多资源，预先获取就可能会给系统带来比较长的延迟。如果资源不是在系统启动时就获取，而是稍后获取，还是会有相应的开销。

已知应用（Known Uses）

预先编译[Hope02] [NewM04]。Java虚拟机经常用预先编译来避免执行时的编译开销。

Pooling模式。使用Pooling模式的解决方案，比如连接池或者线程池通常会预先获取一些资源，比如网络连接或者线程，来很快地响应一开始的请求。

应用服务器[Sun04b]。通常而言，应用服务器的servlet容器不保证servlet何时被加载，也不保证加载的顺序。但是，可以在部署描述信息中为一个servlet指定<load-on-startup>元素，这样容器就会在启动时加载这个servlet。

NodeB [Siem03]。在西门子UMTS基站“NodeB”的软件中，到多个系统部件的连接是在系统启动时预先获取的。这避免了在系统运行时的延迟以及遇到未预测到的资源获取错误。

仓鼠[EvCa01]。一个现实世界中的已知应用是仓鼠（hamster）。它会预先获得尽可能多的果子，然后躲在地洞中吃。仓鼠会把食物存储在颊袋中。

Eclipse插件[IBM04b]。Eclipse是一个通用工具平台，一个开放的可扩展IDE。它的扩展性基于一个允许任意用户成为插件贡献者的插件构架。插件声明决定了插件的可视化特性是一开始就加载的，而包含在JAR中的实际逻辑则是延迟加载的，只有在第一次使用插件的功能时才会加载。

又见（See Also）

与Eager Acquisition模式相反的是Lazy Acquisition模式，这个模式在资源实际被用到时即时分配资源。

Pooling模式把Eager Acquisition模式和Lazy Acquisition模式的优点结合进了一个模式。

Caching模式可以用于管理预先获取的资源。

致谢（Credits）

感谢西门子公司技术部的模式小组，感谢EuroPLoP 2002审稿人Alejandra Garrido，感谢参加笔者的研讨会的Eduardo Fernandez、Titos Saridakis、Peter Sommerlad和Egon Wuchner的宝贵见解和反馈。此外，我们还要感谢Andrey Nechypurenko和Kevlin Henney 对这个模式的源代码示例提出的建议。

2.4 Partial Acquisition模式

Partial Acquisition（部分获取）模式描述了如何把资源获取分成多个阶段来优化资源管理。每个阶段都获取资源的一部分，这取决于系统的约束，比如内存以及其他资源的可用性。

实例（Example）

考虑一个负责管理多个网络元素的网络管理系统（见图2-17）。这些网络元素通常用拓扑树来表示。一棵拓扑树提供了网络基础构架的关键元素的虚拟层次表示。网络管理系统允许使用者查看这棵树并获取关于一个或多个网络元素的细节。取决于网络元素的类型，细节可能会对应大量数据。例如，复杂的网络元素的细节可能包含它的状态以及它的组件的状态。

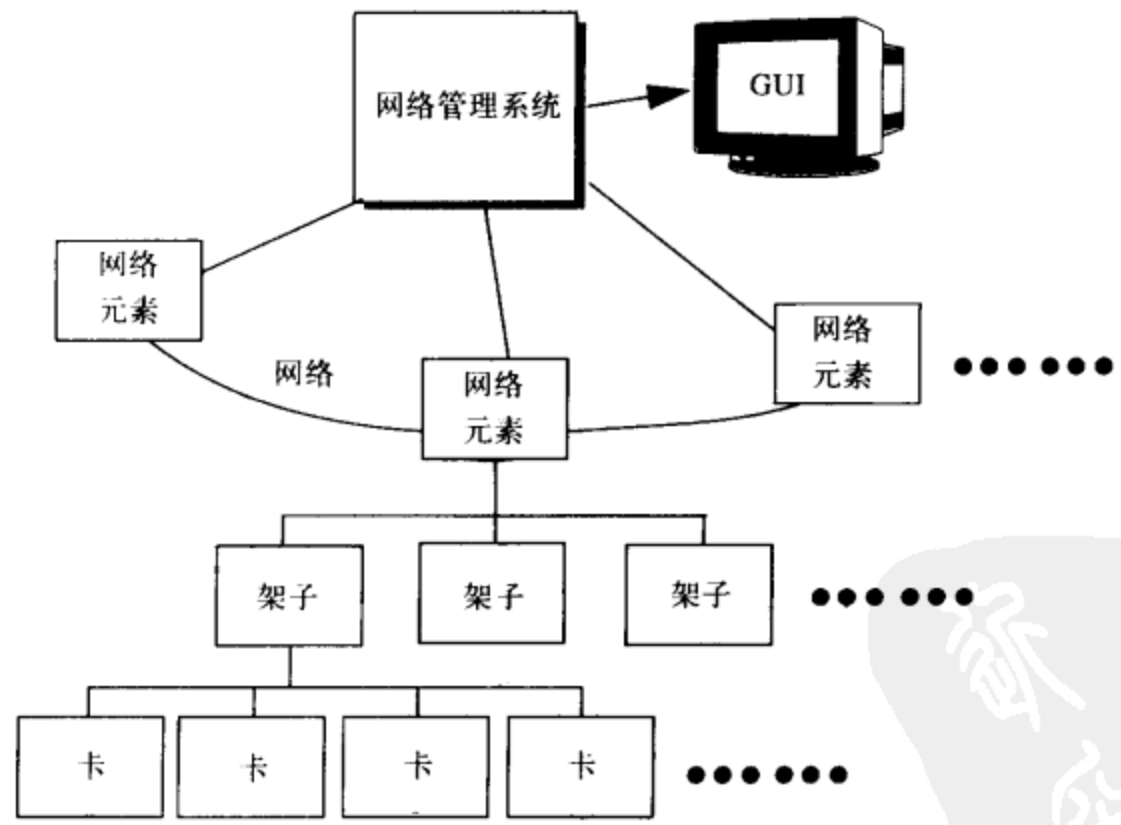


图 2-17

拓扑树通常在应用程序启动时创建，或者在应用程序重启并从错误中恢复时创建。在第一种情况下，所有网络元素的细节，包括它们的组件和子组件，通常都是从物理网络元素获得的。在第二种情况下，信息可以从持久存储器中获得，也可以从物理网络元素获得。但是，无论何种情况，获得所有信息都会给系统启动或者恢复所需的时间带来很大影响。这是因为完整地创建或者恢复网络元素需要创建或者恢复它的所有组件。而且，因为可能每个组件都由多个子组

件构成，创建或者恢复一个组件可能会导致创建或者恢复所有的的子组件。因此，最后生成的层次拓扑树的尺寸以及用来创建或者恢复所有元素的时间可能会难以预期。

背景（Context）

需要高效获取资源的系统。资源的特征是，或者很大，或者尺寸未知。

问题（Problem）

具有高健壮性和可伸缩性的系统必须高效地获取资源，包括本地和远程的资源。预先获取资源对于满足资源的可获得性和可访问性限制很重要，但如果这些系统都在一开始就获取全部资源，那么就会带来很大的额外开销，会浪费很多资源。而另一方面，延迟获取所有资源也是不现实的，因为有的资源可能在应用系统启动或者恢复的时候就要用到。为了解决这些冲突的资源获取需求，我们需要解决下面的作用力：

- 可用性（Availability）。资源获取应当受参数（比如可用的系统内存、CPU负载、其他资源的可用性）的影响。
- 灵活性（Flexibility）。解决方案应当对固定尺寸的资源和未知或者不可预测尺寸的资源工作得同样好。
- 可伸缩性（Scalability）。解决方案应当对资源的尺寸具有可伸缩性。
- 性能（Performance）。资源的获取应当对系统性能的影响尽可能小。

解决方案（Solution）

把资源获取分成两步或者更多步。在每一步中，获取资源的一部分。每步获取资源的数目都应当用一个或多个策略来配置。例如，每步获取的资源的数目可以取决于考虑到可用缓冲区、系统响应时间需求以及所依赖的资源的可用性的策略。当一个资源被部分获取时，资源使用者就可以开始使用它，假定使用之前不需要获得完整的资源。

可以用Eager Acquisition和Lazy Acquisition之类的模式来决定何时执行部分获取资源的一步或者多步。Partial Acquisition模式则决定了资源应该分多少步获取，还决定了每一步获取的资源的比例。

结构（Structure）

下列参与者形成了Partial Acquisition模式的结构：

- 资源使用者获取并使用资源。
- 资源是一个实体（比如影音/视频内容）。资源被分多步获取。
- 资源提供者管理并提供多个资源。

下面的CRC卡片描述了参与者的职责与协作（见图2-18）。

下面的类图展现了参与者之间的依赖关系（见图2-19）。

动态（Dynamics）

场景1：下面的顺序图展示了资源使用者如何以一系列的步骤来部分获取（partially acquire）资源，如图2-20所示。当获取了所有部分之后，它即访问资源。

Class Resource User	Collaborator • Resource
Responsibility • 获取并使用资源	

Class Resource	Collaborator
Responsibility • 从资源提供者被获取，并被资源使用者使用	

Class Resource Provider	Collaborator • Resource
Responsibility • 管理资源并向资源使用者提供资源	

图 2-18

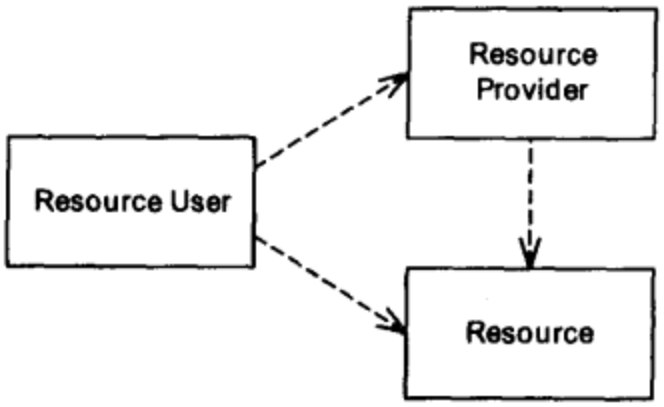


图 2-19

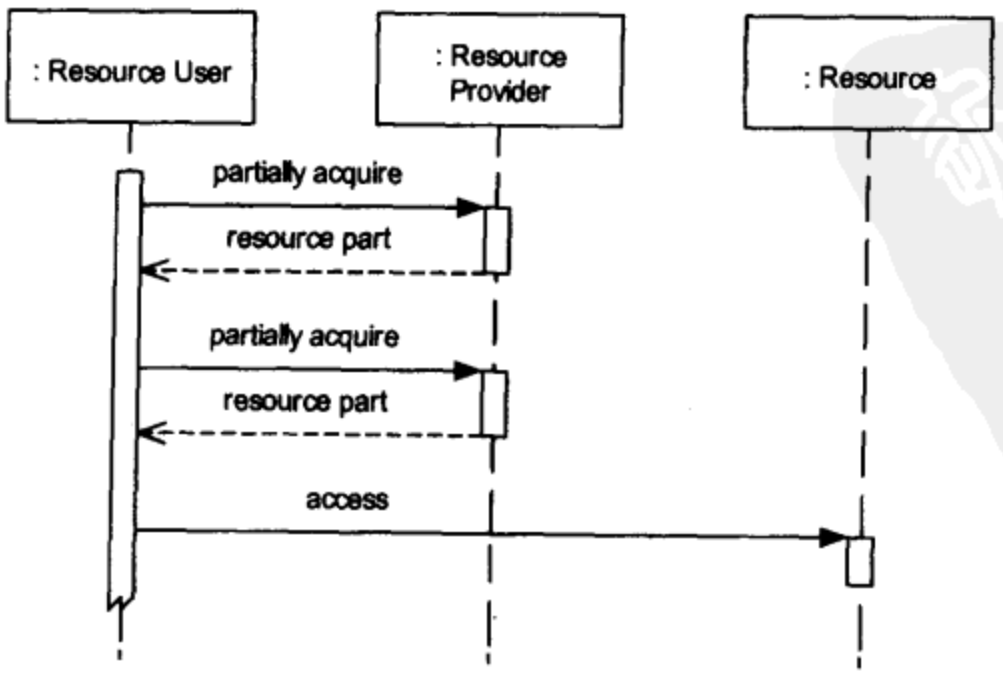


图 2-20

场景2：在某些情况下，资源使用者可能并不知道资源的下一部分是否已经可以获取。例如，资源可能会被增量式地创建，消息流中的网络包的抵达就是一个例子。在这样的情况下，使用者可能不想阻塞于获取操作，而是希望当发生这类事件（event）时获得通知。Reactor模式[POSA2]对这样的场景很有用。这里展示了它是如何动态工作的（见图2-21）。

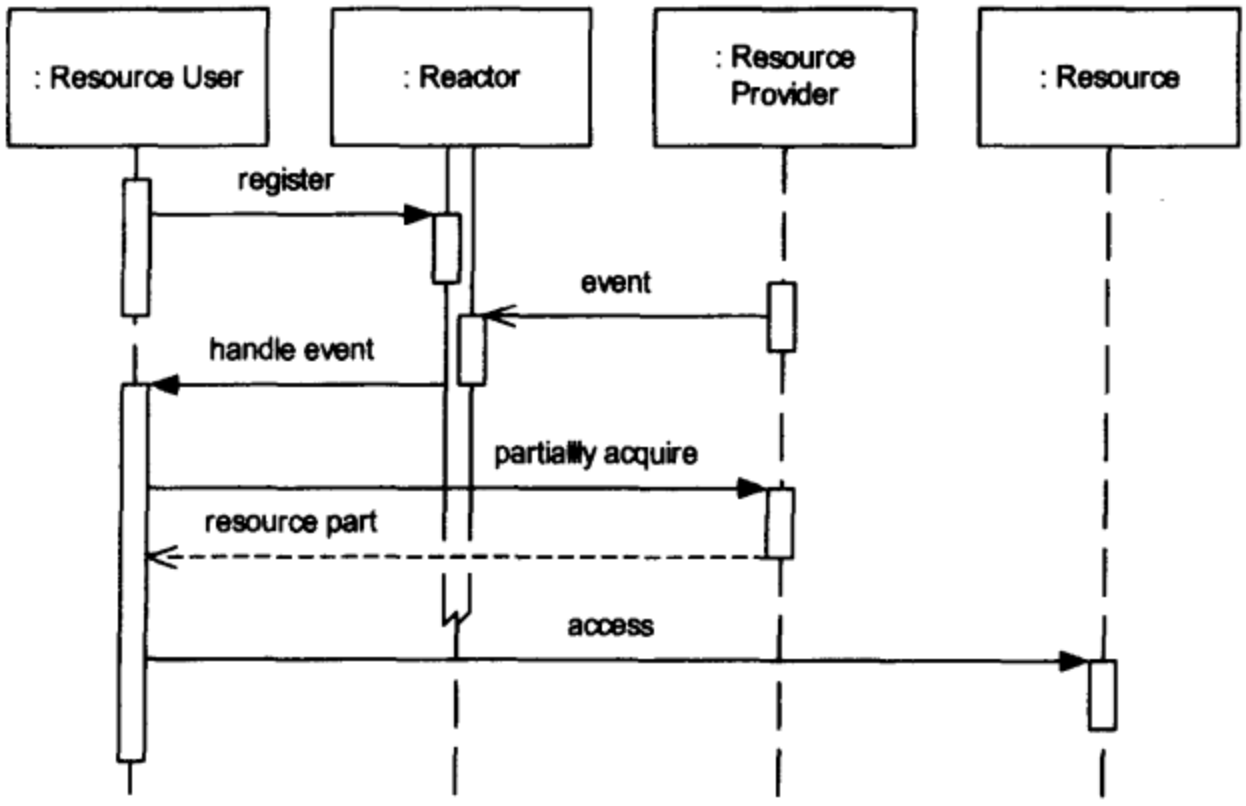


图 2-21

资源提供者的某些关于资源（部分）可用性的事件激发了Reactor，于是Reactor把事件分发给资源使用者，然后资源使用者执行部分获取。更多细节请参见[POSA2]。

实现（Implementation）

实现Partial Acquisition模式涉及六步：

1）*决定步骤的数目*。资源应该分多少步获取取决于系统约束，比如可用的内存和CPU，以及其他的因素，比如时间限制和所依赖的资源的可用性。对于未知或者不可预测尺寸的资源，可能无法决定要分几步来获得整个资源。在这种情况下，步骤数没有上限，会一步步执行下去直到完整地获得资源。

在网络管理系统的例子中，步骤数目可以同拓扑树的层次数目相对应。每一步都可以通过获得相应层次的组件的细节来创建一个完整的层次。如果层次结构中的某一层非常复杂，包含了大量组件，那么获得该层所有组件的细节的操作可以进一步划分成两个或多个步骤。

2）*选择获取策略*。决定何时应该执行资源获取的哪一个步骤。可以用Lazy Acquisition和Eager Acquisition之类的模式来控制何时应执行一个或者多个资源获取步骤。例如，可以用Eager Acquisition模式来获取资源的起始部分，然后用Lazy Acquisition模式来获取资源的剩余部分，或者在某个系统启动完成之后但用户还没有请求资源的中间时间来获取。

在网络管理系统的例子中，可以用Eager Acquisition模式来获取网络元素的细节，但不获取其内部组件的细节。网络元素的组件和子组件的细节可以当用户在GUI中选择了网络元素并试图

查看组件细节时再通过Lazy Acquisition模式来获取。

3) 决定获取多少。配置策略来决定每一步要部分地获取多少资源。可以配置不同的策略来决定每一步获得多少资源。如果资源的尺寸是确定的，那么可以用简单的策略，平均分配每一步所获得的资源。更复杂的策略会考虑可用的系统资源。因此，例如，当具有足够内存的时候，这样的策略可能会在一步中获得较多部分的资源。稍后，当系统资源不足的时候，这样的策略可能会获取较少部分的资源。

如果要获取的资源的尺寸未知或者不可预测，那么可以用这样的自适应策略。还可以配置其他的策略来利用其他的参数，比如需要的相应时间。如果没有系统约束，那么另一个策略可以贪心地尽可能多地获取资源。这样的策略可以保证整个资源在可能的最短时间内被获得。为了判断使用何种策略比较合适，有必要良好地理解应用程序的语义。

4) 引入缓存（可选项）。决定是否要把部分获取的资源缓存起来。如果资源尺寸未知，或者整个资源使用前需要在某处合并，那么缓存资源是有用的。如果资源需要被缓存，那么需要决定分配多大空间的缓存区，以确保可以装入整个资源。对未知或者不确定大小的资源，分配的缓冲区尺寸应该在系统限制（比如可用内存）之内，但又要足够大，以处理系统中资源尺寸的上限。

5) 实现获取触发器。建立一个负责执行资源获取的每一步的机制。这样的机制应当负责用多个步骤来获取资源的不同部分。Reactor [POSA2]之类的模式可以用于实现这样的机制。例如，Reactor可以用于在资源的一部分变得可用的时候获取它们。也可以建立另一种机制，前摄地[POSA2]获取资源的各个部分。

6) 处理错误条件和部分失败。错误条件和部分失败是分布式系统的特性。当使用Partial Acquisition模式时，有可能会在一步或者多步结束之后出错。结果可能是获取了一部分资源，但是试图接着获取资源的剩下部分会失败。取决于应用程序的语义，这样的部分失败可能可以结束，也可能不可以接受。比如，如果分多步获取的资源是文件的内容，那么部分失败会使得已经成功获取的数据变得不一致。

另一方面，在网络管理系统的例子中，未能成功获得某个子组件的细节对已经成功获得的其余组件的细节没有影响。即便部分失败，但成功获得的组件的细节还是可以供用户使用。

处理部分失败的一种方式是使用Coordinator模式。这个模式有助于确保资源获取的所有步骤都成功完成，或者确保一步都没有成功。

实例解析（Example Resolved）

思考一下网络管理系统，网络元素本身内部包含多个组件，比如CPU板、连接交换器、内存。加载这些组件的细节可能会很费时间。用Partial Acquisition模式，可以把获取网络元素以及它们的组件的细节的任务分成多个步骤。在第一步，只从物理网络和数据库中获取网络元素的细节，并不获取组件的细节。

网络管理系统的拓扑管理器向可视化子系统提供网络元素及其组件的细节，以显示它们。拓扑管理器从数据库或者从物理网络元素获得信息（取决于数据的类型，分别为静态配置或者动态操作参数）。下面的Java代码展示了TopologyManager如何把获取网络元素的组件的细节延迟到后面的步骤。

```
public class TopologyManager
{
   // Retrieves the details for a specific network element.

   public Details getDetailsForNE (String neId) {
       Details details = new Details();

       // Fetch NE details either from physical network
       // element or from database ...

       // ... but defer fetching details of NE subcomponents
       // until later. Create a request to be inserted into
       // the Active Object message queue by the Scheduler.

       FetchNEComponents request =
           new FetchNEComponents (neId);

       // Now insert the request into the Scheduler so that
       // it can be processed at a later stage.
       scheduler.insert (request);

       return details;
   }
   private Scheduler scheduler;
}
```

实际获取网络元素组件的操作是由Active Object [POSA2]异步完成的。对此，请求被Scheduler排队，Scheduler运行于Active Object的线程。当被Scheduler激发后，请求会获取网络元素的组件。

```
public class Scheduler implements Runnable {

   public Scheduler (int numThreads, ThreadManager tm) {
       // Spawn off numThreads to be managed by
       // ThreadManager. The threads would dequeue
       // MQRequest objects from the message queue and
       // invoke the call() method on each one
       // ...
   }

   // Insert request into Message Queue
   public void insert (Request request) {
       queue.insert (request);
   }

   public void run() {
       // Run the event loop
       // ...
   }

   private MessageQueue queue;
}
```

获取网络元素的组件的细节的主要逻辑包含于FetchNEComponents类。它实现了Request接口，所以可以被Scheduler调度。

```
public interface Request {
   public void call ();
   public boolean canRun ();
}
```

Scheduler会调用canRun() 来检查是否可以执行Request。如果返回值为真，它会调用Call()，否则重新调度请求。

```
public class FetchNEComponents implements Request {
   public FetchNEComponents (String neId) {
       // Cache necessary information
   }

   // Hook method that gets called by the Scheduler.
   public void call () {

       // Fetch NE subcomponents using Partial Acquisition
   }

   // Hook method that gets called by the Scheduler to
   // determine if this request is ready to be processed.
   public boolean canRun () {
       // ...
       return true;
   }
}
```

类之间的交互在图2-22的顺序图中描述。当用户选择一个网络元素时，拓扑管理器（Topology Manager）会创建一个请求，它会一步步地获取架子和卡片信息。每一步获取的信息都被传递给用户界面（User Interface）以可视化。当用户选择了一个架子的相同网络元素时，细节会被很快返回，因为它们在后台被预先获得了。

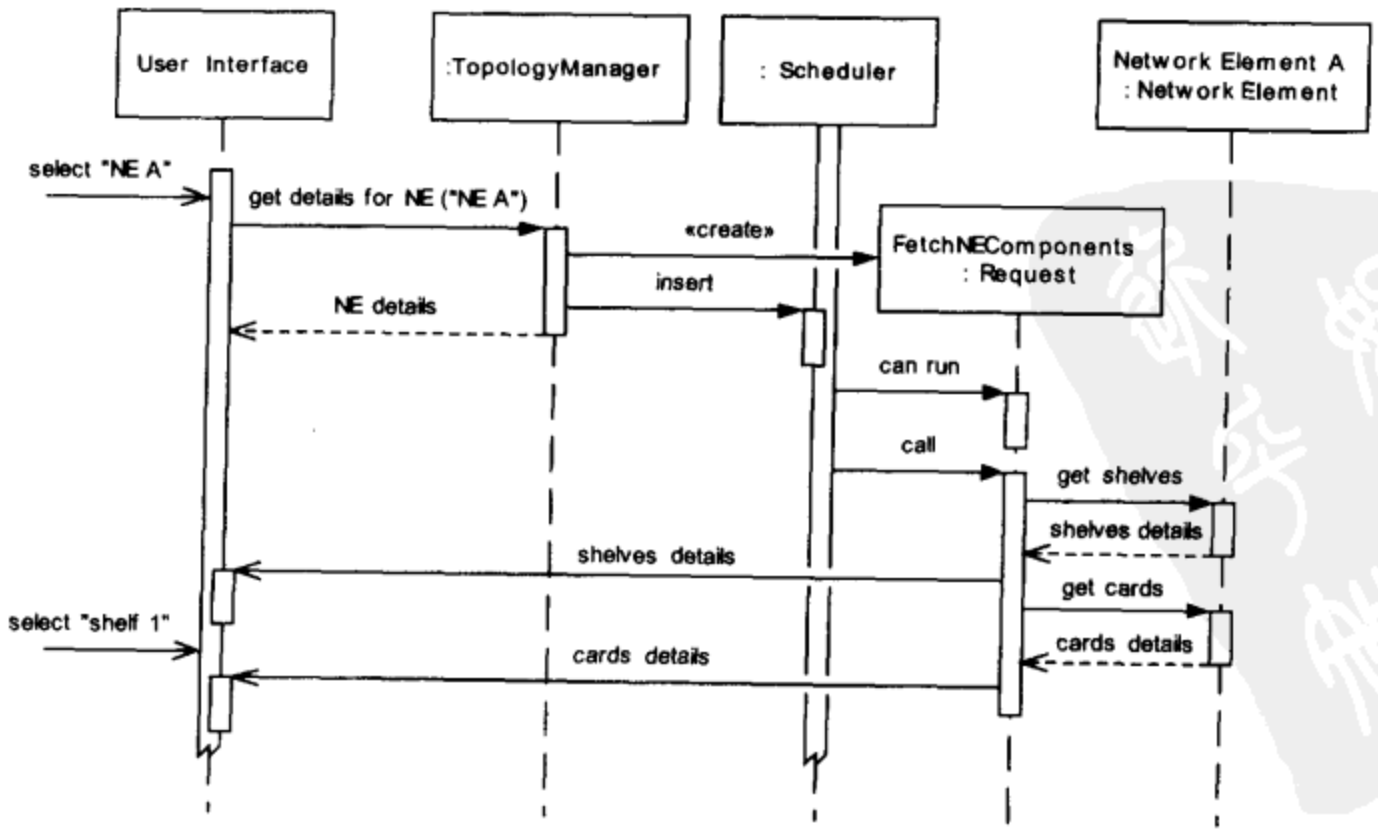


图 2-22

使用Partial Acquisition模式，拓扑树是分多步创建的。结果是访问网络元素时性能有了显著改善。第一次请求网络元素细节（通常是在启动时进行）会触发对网络元素的组件的细节的部分获取，于是当用户请求它们时这些细节即时可用。此外，还可以增加额外的逻辑来使得对网络元素细节的部分获取可以触发对邻近网络元素的获取。为了透明地整合Partial Acquisition模式，可以使用Interceptor模式[POSA2]（参见“变体（Variants）”一节）。

变体（Variants）

Transparent Partial Acquisition模式。可以引入一个interceptor用于截获资源使用者的资源获取请求，并且可以用配置策略获取资源的初始部分。然后在接下来的步骤中以对资源使用者透明的方式获得剩下的资源部分。例如，在前面的例子中，可以用interceptor来截获请求，并且只获取网络元素。interceptor不会立即获取子组件，因为可以稍后以对资源使用者透明的方式获得它们。

结果（Consequences）

使用Partial Acquisition模式有几个**优点**：

- *反应式行为*（Reactive behavior）。Partial Acquisition模式使我们可以获取缓慢地变得可用或者局部可用的资源。如果不使用Partial Acquisition模式，资源使用者就不得不等待不定长的时间以等待整个资源变得可用。
- *可伸缩性*（Scalability）。Partial Acquisition模式允许要获取的资源的尺寸具有可伸缩性。资源获取的步骤数目可以依据要获取的资源的尺寸来配置。
- *可配置性*（Configurability）。Partial Acquisition模式可以用一个或多个决定分多少步获取资源以及一步获取多少资源的策略来配置。

使用Partial Acquisition模式有几个**缺点**：

- *复杂性*（Complexity）。用户处理资源的算法需要可以处理只获得了一部分的资源。这可能会给应用程序带来一定程度的复杂性。此外，用Partial Acquisition模式会导致错误处理变得更为复杂。如果某一步失败了，那么错误处理策略必须确保完整的动作被重新执行或者更正。另一方面，错误处理也会变得更健壮。如果某一步获取失败，那么可以重新执行同一步而不需要重新进行全部获取操作。
- *额外开销*（Overhead）。Partial Acquisition模式要求资源分步获取。这可能会带来额外的调用开销（多次调用以获得相同资源的不同部分）。

已知应用（Known Uses）

渐进的图像加载。大多数现代的Web浏览器（比如Netscape [Nets04]、Internet Explorer [Micr04]或者Mozilla [Mozi04]）都实现了Partial Acquisition模式，渐进地加载图像。浏览器首先下载Web页面的文字内容，同时为页面上将要显示的图像建立标记。然后，浏览器渐进地下载和显示图像，与此同时用户可以阅读页面的文本内容。

Socket输入[Stev03]。从socket读取数据通常也会使用Partial Acquisition模式。因为通常完整的数据不会经由socket一次传入并可用，所以需要进行多次读操作。每次读操作都从socket获得一

部分数据，并把它存储于某个缓冲区。一旦所有的读操作完成了，缓冲区就包含了最终的结果。

数据驱动的遵循协议的应用程序。数据驱动并遵循某个特定协议的应用程序也使用了Partial Acquisition模式。通常，这样的程序会遵从特定的协议来分两步或者多步获取数据。例如，处理CORBA IIOP协议（Internet Inter-ORB Protocol）[OMG04a]请求的应用程序通常会在第一步读取IIOP头，以确定请求体的尺寸。然后，它会一步或者分多步读取请求体的内容。所以，这样的应用程序会使用部分获得的资源。在应用程序处理IIOP请求的例子中，程序使用了在第一步获得的IIOP头。

堆压缩。[GKVI+03]的研究涉及Java语言的一个特殊的垃圾收集器，使得使用小于应用程序实际尺寸的堆成为可能，这是通过临时性地压缩堆中不用的对象来做到的。当再次访问被压缩的对象时，只有对象的一部分会被解压缩，这导致了内存的部分分配。如果需要用到对象的其余部分，那么会在后面的步骤中解压缩。

声音/视频流[Aust02]。当对声音/视频流解码时，流是一部分一部分地获取的。对于视频，算法通常会一次获取一帧或多帧，然后解码、缓冲、显示。

又见（See Also）

Reactor模式[POSA2]是用来高效地分派事件的。在I/O操作的情况下，它可以反应式地提供多个用于局部获取的资源。可以使用缓存来保留每一步中部分获得的资源。这对于正在获取的尺寸未知的资源特别有用。

致谢（Credits）

感谢西门子公司技术部的模式小组，感谢PLoP 2002审稿人Terry Terunobu，感谢参加笔者的研讨会的Angelo Corsaro、Joseph K. Cross、Christopher D. Gill、Joseph P. Loyall、Douglas C. Schmidt和Lonnie R. Welch的宝贵见解和反馈。

第 3 章
资源生命周期

“我的灵魂啊，不要再去苦苦寻求圣哲的生活了，淋漓尽致地享受能拥有的一切吧。”

——Pindar

概览

一旦成功获取到了资源，就需要有效地管理资源的生命周期了。这包括使用户可以访问资源，处理资源之间的依赖，获取被依赖的资源（如果有必要的话），以及释放不再需要的资源。

Caching模式描述了如何管理频繁访问的资源的生命周期以降低重新获取和释放这些资源的开销，同时维护了资源的标识。这是一个常用的模式，很多具有高度可延展性的企业解决方案都使用了这个模式。同Caching模式相比，Pooling模式优化了资源的获取和释放，但不维护资源的标识。因此，对无状态资源而言，Pooling模式更合适，因为它们几乎不需要初始化。同Caching模式类似，Pooling模式也被广泛使用，例子包括组件平台中的组件池以及分布式应用中的线程池。Caching模式和Pooling模式都只适用于可重用的资源。两个模式通常都只应用于被用户顺序使用的排它型可重用资源。但是，在一些情况下，对可同步访问的可重用资源使用Caching模式或者Pooling模式也是合理的。在这样的情况下，Caching模式和Pooling模式并不知道资源是否被同步访问，因为只有从缓存或者池中获得资源后才会进行资源访问。

两个或者多个实体（比如获得的资源、资源使用者或者资源提供者）可以相互交互并对软件系统造成变化。在这样的情形下，我们认为这些实体是活动的，能够参与交互，可以导致变化。考虑到这些实体，要求产生的所有改变都保证系统处于一致的状态是很重要的。Coordinator模式确保在涉及多个实体的任务中，系统状态保持一致，从而维持了整体的稳定性。

Resource Lifecycle Manager模式管理软件系统的所有资源，从而免除了被管理的资源以及资源使用者的管理责任。Resource Lifecycle Manager负责管理所有类型的资源的生命周期，包括可重用资源和不可重用资源。

3.1 Caching模式

Caching（缓存）模式描述了如何通过用完资源后不立刻释放资源来避免对资源的昂贵的重新获取。资源会维持它们的标识，并保留在某种可以快速访问的存储器中。它们可以被重用，从而避免了重新获取。

实例（Example）

考虑一个网络管理系统（见图3-1），它需要监视很多网络元素的状态。网络管理系统通常被实现为3层系统。最终用户使用表示层来和系统交互，表示层通常是一个图形用户界面。中间层

则包含了业务逻辑，同持久层交互，并且也负责同物理网络元素通信。因为典型的网络可能会包含上千的网络元素，所以在中间层、应用服务器和所有网络元素之间建立起持久连接的代价非常高昂。另一方面，最终用户可以通过图形用户界面选择任意网络元素，以便获取该网络元素的细节。网络管理系统必须对用户请求可以快速响应，所以在用户选择网络元素和网络元素的属性显示出来之间的延迟应当比较小。

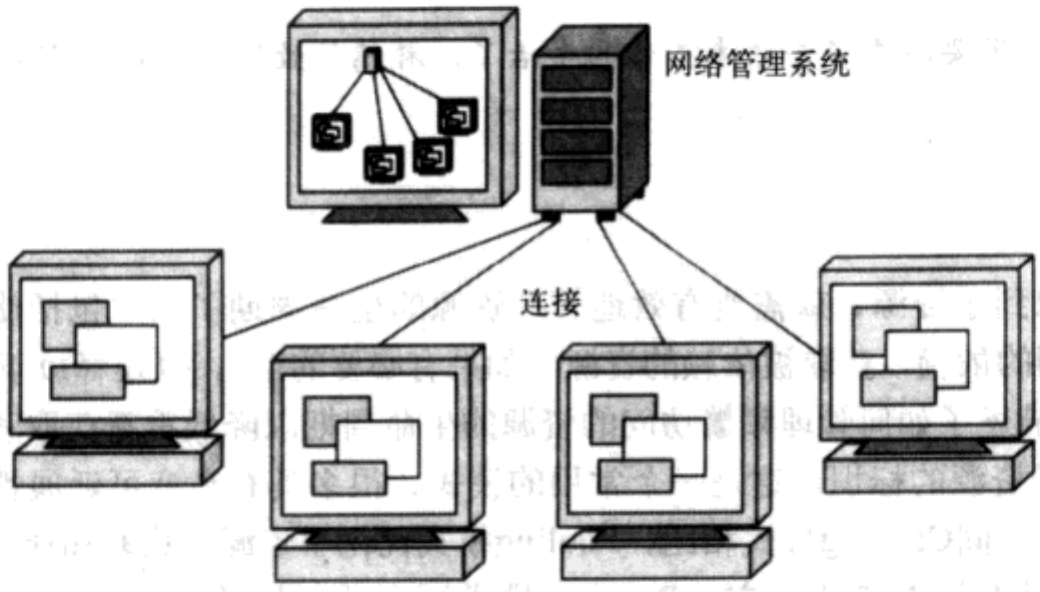


图 3-1

为每个用户选择的网络元素建立一个新的网络连接，然后在使用之后释放这个连接，这会带来很大的开销（表现为服务器内部占用的CPU周期）。此外，访问网络元素的平均时间也可能会太长。

背景（Context）

反复访问同一组资源的系统，需要为性能而优化。

问题（Problem）

对相同资源的重复的获取、初始化以及释放造成了不必要的开销。在系统的同一个组件或者多个组件访问相同资源的情形下，重复的获取和初始化带来了CPU周期和系统整体性能的开销。应该减少获取、访问和释放频繁使用的资源的开销，以提高性能。

为了解决这个问题，需要解决这些作用力：

- 性能（Performance）。重复获取、初始化和释放资源的开销必须最小化。
- 复杂性（Complexity）。解决方案不应当使得获取和释放资源的操作变得更复杂、更麻烦。此外，解决方案也不应对资源访问增加不必要的间接层。
- 可用性（Availability）。解决方案应当在资源提供者暂时不可用的情况下也使得资源可以被访问。
- 可伸缩性（Scalability）。解决方案应该对于资源的数目而言是可伸缩的。

解决方案（Solution）

暂时把资源存储在称做“缓存”的快速访问缓冲区中。当资源后来被再次访问时，用缓存

来获取和返回资源，而不需要从资源提供者（比如管理资源的操作系统）重新获取它。缓存通过资源标识（比如指针、引用、主键）来识别资源。

保持频繁访问的资源，不释放它们，这避免了重新获取和释放资源的开销。使用缓存简化了对访问资源的组件的管理。

当缓存中的资源不再需要时，它们就被释放了。缓存的实现决定了如何去除不再需要的资源。这一行为受策略控制。

结构（Structure）

下列参与者形成了Caching模式的结构:

- 资源使用者使用资源。
- 资源是一个实体（比如连接）。
- 资源缓存（Resource Cache）缓冲了资源使用者释放的资源。
- 资源提供者拥有并管理多种资源。

下面的CRC卡片展示了参与者如何互相交互（见图3-2）。

Class **Resource User**	*Collaborator*
Responsibility	• Resource • Resource Provider • Resource Cache
• 开始从资源提供者处获取资源 • 使用资源 • 把不用的资源释放到缓存中 • 从资源缓存获取被缓存的资源	

Class **Resource**	*Collaborator*
Responsibility	
• 被从资源提供者处获得、并被资源使用者使用	

Class **Resource Cache**	*Collaborator*
Responsibility	• Resource • Resource Provider
• 缓冲资源 • 最终清除资源	

Class **Resource Provider**	*Collaborator*
Responsibility	• Resource
• 拥有并管理多个资源	

图 3-2

下面的类图描述了Caching模式的结构（见图3-3）。

资源提供者（比如操作系统）管理资源提供者已开始获得的资源。然后资源使用者访问资源。当资源不再需要时，就释放到缓存中。资源使用者使用缓存来获得需要重新访问的资源。与从资源提供者获取资源相比，从缓存中获取资源的代价比较低（更少的CPU占用、更小的延迟）。

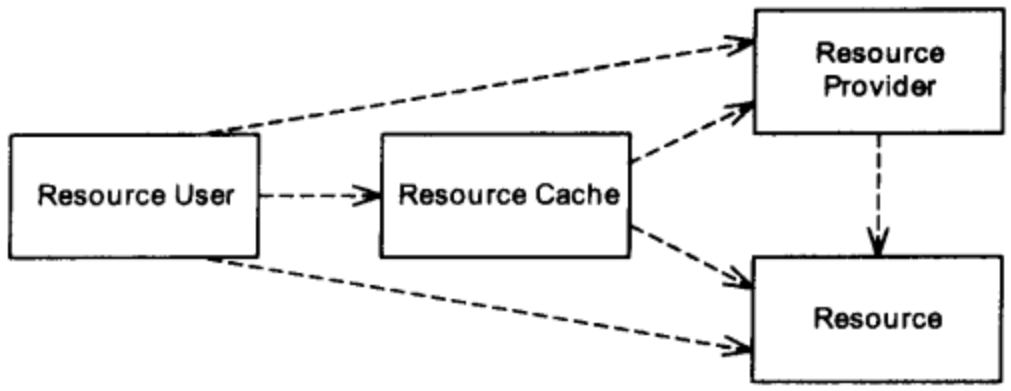


图 3-3

动态（Dynamics）

图3-4展示了资源使用者如何从资源提供者获取资源。然后，资源被使用者访问。资源使用过后放入缓存，而不是释放（release）给资源提供者。

当资源使用者需要再次访问相同的资源时，就使用缓存来获取它。

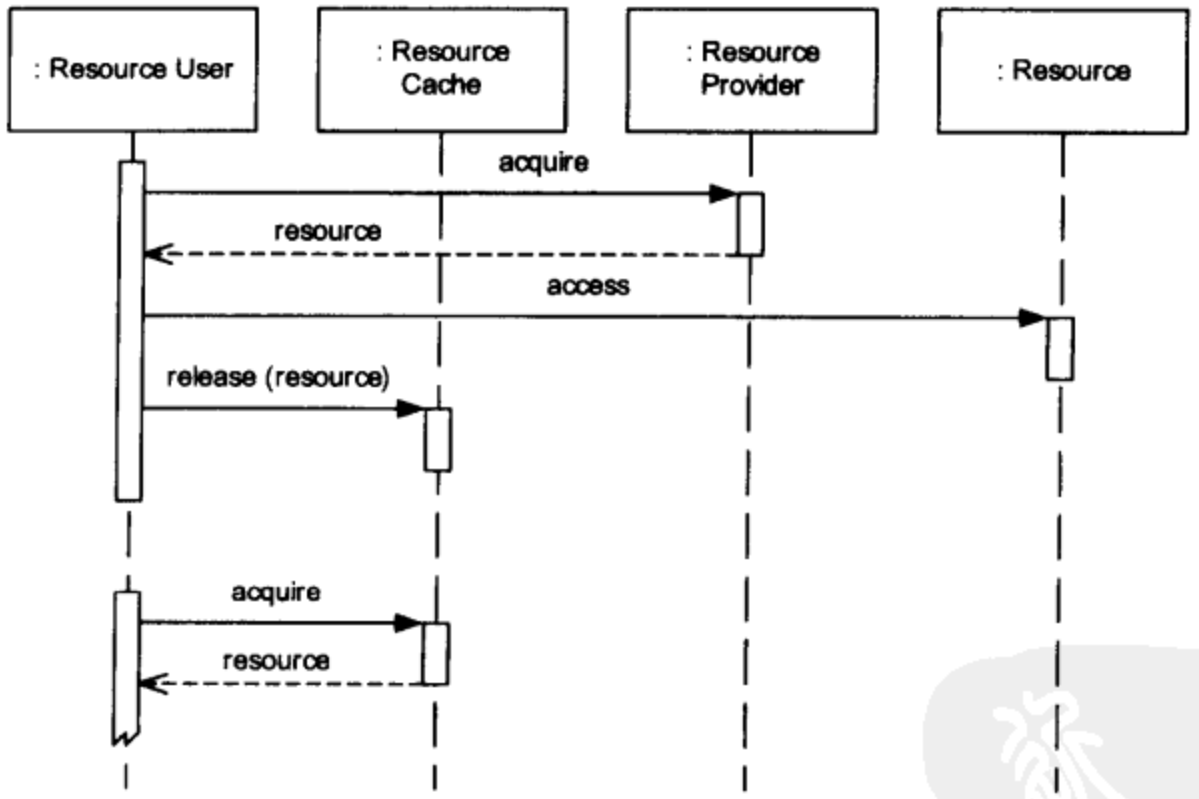


图 3-4

实现（Implementation）

要实现Caching模式，需要进行下列步骤：

1）选择资源。选择可以从缓存获益的资源。这些通常是获取代价高昂、使用频繁的资源。缓存经常在发现性能瓶颈之后作为优化手段被引入。

在分布式系统中，存在两种缓存：客户端缓存和服务器端缓存。客户端的缓存对于节省带宽以及节省用于重复地从服务端传送数据到客户端的时间很有效。而服务端缓存则有助于解决多个客户请求导致重复获取和释放服务端的相同资源的问题。

2) 决定缓存接口。资源若要被释放，并被资源使用者直接从缓存中重新获取，就必须设计一个合适的接口。这个接口需要提供release()和acquire()方法。

```
public interface Cache {
   public void release (Resource resource);
   public Resource acquire (Identity id);
}
```

上面的接口依赖于独立的资源标识。但并不是所有的资源都有独立的标识。在某些情况下，资源标识可能需要从资源的属性计算得出。⊖

资源使用者把资源释放到缓存（而不是释放给资源提供者）时会调用release()方法。

3) 实现缓存。Cache接口中acquire()和release()方法的实现提供了缓存的主要功能。

下面的代码片段展示了资源使用者在释放资源时调用的release()方法的实现。

```
public class CacheImpl implements Cache {
   public void release (Resource resource) {
       String id = resource.getId ();
       map.put (id, resource);
   }
   // ...

   HashMap map;
}
```

release() 方法把资源添加到map中，这样稍后对acquire()的调用就可以通过它的标识找到资源。为了达到优化的目的，建议使用哈希表，因为哈希表可以执行几乎是常数时间复杂度的查找。Comparand模式[CoHa01]提供了一些如何在标识间进行比较的想法。取决于资源的类型，需要首先判定资源的标识。在我们的例子中，资源可以识别自身。

缓存实现中的acquire() 方法应当负责基于标识从表中查找资源。另一种变体是，当资源从缓存中获取失败时，那就意味着没有找到所需标识的资源，那么缓存本身可以从资源提供者获取资源。更多的细节请参见“变体（Variants）”一节的Transparent Cache这一变体。

下面的代码片段展示了acquire() 方法的实现。

```
public class CacheImpl implements Cache {
   public Resource acquire (Identity id) {
       Resource resource = map.get (id);
       if (resource == null)
           throw new ResourceNotFound ();
       return resource;
   }
}
```

4) 决定如何整合缓存（可选）。如果必须透明地集成缓存，就可以使用Interceptor模式[POSA2]或者Cache Proxy模式[POSA1]。引入Interceptor或者Cache Proxy把操作变得透明，从而降低了显式地释放资源到缓存以及从缓存重新获取资源的复杂性。关于如何把操作变得透明，请参见“变体（Variants）”一节中的Transparent Cache这一变体。不过，使用这样的方法无法避

⊖ 我们不讨论这个问题，因为这超出了本书的范围。

免在缓存中查找所带来的间接层次。

5) 决定清除策略。存储在缓存中的资源会占用存储空间。若许久未用，那么保留这些资源就变得低效了。因此，应该使用Evictor模式来删除不再需要使用的资源。可以用多种方式来整合Evictor模式。例如，可以在release()方法中调用Evictor，也可以通过定时器以一定的间隔来执行。当然，这样的行为会影响整体的可预测性。此外，可以用不同的策略来配置Evictor，比如Least Recently Used（LRU）、Least Frequently Used（LFU），以及其他的领域相关策略。可以使用Strategy模式[GoF95]。

6) 确保一致性。很多资源都有相关联的状态，必须在资源创建时正确地初始化这些状态。此外，当资源被写操作所访问时，需要在原来的资源和镜像原来资源的缓存中的资源之间保持一致性。当原来的资源发生了改变，就需要通过回调函数来告诉镜像更新它的拷贝。若镜像发生了改变，大多数缓存使用称做“写通”的策略。使用这一策略，对镜像的改变会直接作用于原来的资源以及镜像的资源。这一功能通常由称做Synchronizer的实体来实现。因此，Synchronizer成了这个模式的重要参与者。有些缓存则进一步地优化了这项功能，引入了更复杂的逻辑以确保原来的资源和缓存中的镜像保持一致。

用Strategy模式[GoF95]来决定何时需要同步。在某些情况下，只有特殊的操作（比如写操作）需要立即同步，而在其他的情况下可能定期更新更合适。另外，也可以通过外部的事件来触发同步，比如其他资源使用者更新了原来的资源。

在前面的例子中，如果网络元素发生了改变，那么被缓存的网络元素的内存表示也必须改变。类似地，如果用户更改了一个网络元素的设置，那么改动必须反映到物理的网络元素。

实例解析（Example Resolved）

考虑需要监视很多网络元素的状态的网络管理系统。网络管理系统的中间层会使用Caching模式来实现到网络元素的连接缓存。当用户访问一个特定的网络元素时，到网络元素的连接就被获取。当连接不再需要时，那么就会被加入缓存。稍后，当新的对此连接的请求抵达时，连接就会从缓存中来获取，这样就避免了高的获取开销。

接下来，到其他网络元素的连接会在用户第一次访问它们时建立。当用户把环境切换到另一个网络元素时，那么连接就被放回连接缓存。当用户访问同一个网络元素时，连接就会被重用。第一次访问被重用的连接时不会产生延迟。

变体（Variants）

透明缓存（Transparent Cache）。当缓存必须被透明地整合时，可以用Lazy Acquisition模式来获得客户所请求的资源。这只有在缓存知道如何获取及初始化这样的资源的情况下才可行。通过在一开始延迟获取资源，资源使用者可以不必关心缓存的存在。

预取缓存（Read-Ahead Cache）。在使用重复的Partial Acquisition模式来获取资源的情况下，如果使用预取缓存，那么系统可以设计得更为高效。预取缓存可以在实际使用之前获取资源，确保资源在需要时可用。

缓存池（Cached Pool）。缓存和池的组合可以提供复杂的资源管理解决方案。当资源需要从缓存中被释放时，它可以被放回到池中，而不必返回给资源提供者。从而缓存扮演了资源的中

间存储场所的角色。缓存可以用超时来配置。一旦时间到了，那么资源就失去其标识，进入池中。优势是一个小的优化：如果要求获取相同的资源，该资源还没被返回到池中，那么你避免了初始化的开销（缓存的好处）。

层次缓存（Layered Cache）。在复杂的系统中，缓存模式经常用在多个层次上。例如，Websphere应用服务器（Websphere Application Server，WAS）[IBM04a]就在多个层次运用了缓存。WAS提供了主动缓存EJB方法的返回结果的能力。把命令存储在缓存中，以供后续调用重用，这就使得请求可以在业务层被处理，而不是在数据层，从而避免了在数据层处理的昂贵开销。WAS还提供了一个就绪语句的缓存，可以被后端数据库配置以处理动态/静态的SQL语句。取决于应用程序数据访问模式，WAS的就绪语句缓存可以提升应用程序性能。为了提高JNDI操作的性能，WAS利用缓存来减少到名字服务器的查找操作的远程调用数目。最后，WAS提供了数据访问bean，提供了对数据库查询结果的缓存。

结果（Consequences）

缓存增加了一些性能开销，这是由额外的间接层次导致的。但是总体来说，性能得到了提升，因为资源获取起来更快。

使用Caching模式有以下**优点**：

- *性能*。对频繁使用的资源的快速访问是缓存的一个明显好处。和Pooling模式不同，缓存确保资源维持它们的标识。因此，当相同的资源需要被再次访问时，资源就不需要从别的地方获取，它已经在那里了。
- *可伸缩性*。避免资源获取和释放是缓存的一个“隐藏”的好处。本质上缓存的实现方式是保存频繁访问的资源。因此，和Pooling模式一样，Caching模式有助于避免获取和释放资源的开销。这对于频繁的使用尤其有好处，从而提升了可伸缩性。
- *使用复杂性*。缓存确保了从使用者的角度获取和释放资源的复杂性不会增加，虽然需要有额外的步骤来检查资源是否在缓存中可获得。
- *可用性*。缓存资源在资源提供者暂时不可用的时候增加了资源的可用性，因为被缓存的资源还是可用的。
- *稳定性*。因为Caching模式减少了释放和重新获取资源的操作量，所以降低了内存碎片的可能性，从而会带来更好的系统的稳定性。这和Pooling模式类似。

使用Caching模式也有一些**缺点**：

- *同步复杂性*。取决于资源的类型，复杂性会增加，因为需要保证被缓存的资源和缓存资源所代表的原始数据的状态的一致性。这一复杂性在集群环境中变得更加显著，在某些极端的情况下会使得这个模式毫无用处。
- *持久性*。对被缓存的资源的改变在系统崩溃时可能会丢失。但是，如果使用了同步缓存，那么这个问题可以被避免。
- *空间开销*。系统的运行时空间开销会增加，因为很可能没被使用的资源也被缓存了。但是，如果使用Evictor模式，那么可以把这种未使用的被缓存的资源的数目最小化。

如果应用程序要求访问代价高昂的资源的数据总是可用的，那么缓存不是一个好主意。例如，中断驱动的、有很多I/O操作的应用程序以及嵌入式系统常常没有硬件内存缓存。

对优化的一个一般注解：应当慎重地使用缓存，除非其他的手段（比如优化对资源获取的操作本身）无法更进一步。缓存会带来一些实现复杂性，并增加整体方案的维护难度，还会增加整体的资源消费（比如内存消耗），因为被缓存的资源没有被释放。因此，在使用Caching模式之前，要考虑性能、资源消费和复杂性之间的取舍。

已知应用（Known Uses）

硬件缓存[Smit82]。几乎所有的CPU都有其硬件内存缓存。这个缓存不仅减少了内存访问时间，还有助于减少总线通信量。缓存内存通常会比RAM至少快2倍。

文件系统和分布式文件系统中的缓存[NW088][Smit85]。分布式文件系统在服务器端使用Caching模式来避免始终从磁盘读取文件块。最常用的文件被保留在服务器的内存中，以便进行快速访问。文件系统还在客户端使用缓存来避免不得不从服务器获取数据而造成的网络通信和延迟。

数据传输对象[Fowl02]。CORBA [OMG04a]和Java RMI [Sun04g]之类的中间件技术使得对象可以进行远程传输，而不只是对远程对象执行远程方法调用。当它们的方法在本地被调用时，远程对象在客户端和服务端之间其实是按值传输的。这是为了把昂贵的远程方法调用最小化。对象在本地被保存在一个缓存中，并且代表了实际的远程对象。虽然这个方法提高了性能，但是用户必须实现对象的本地拷贝和远程原本之间的同步。

Web代理[Squi04]。Web代理是一个代理服务器，介于Web浏览器和Web服务器之间。每次Web浏览器从万维网的任一Web服务器上访问页面时总会访问Web代理。Web代理会把被浏览器访问的页面缓存下来，并在收到对同一页面的后续请求时返回缓存的页面。结果就是，Web代理通过在本地缓存中保存频繁请求的页面从而减少了在因特网上传递的Web页面请求的数量。

Web浏览器。大多数流行的Web浏览器（比如Netscape [Nets04]）和Internet Explorer [Micr04]都会缓存频繁访问的Web页面。如果用户访问相同的页面，那么浏览器就从缓存中获取页面内容，从而避免了在Web站点上获取内容的昂贵操作。浏览器会用时间戳来决定把页面在缓存中保留多久，什么时候清除它们。

换页[Tane01] [NoWe00]。现代操作系统把页面保留在内存中以避免从磁盘上的交换空间读取它们的昂贵操作。这些保留在内存中的页面可以被认为是保留在缓存中。只有页面没有在缓存中找到时，操作系统才会从磁盘上去获取它。

文件缓存[KiSa92]。文件缓存提升了性能，并且使得装载的网络驱动器的文件和目录在离线时可用，而不必建立网络连接。文件和目录会被缓存，并且当连上网络后会和原来的数据进行同步。微软操作系统的最新版本通过一个称为“Offline Files”的功能支持文件缓存。

.NET[Rich02]。.NET的数据集可以看做是在内存中的数据库。它们在本地被实例化，使用SqlDataAdaptor，通过对数据库的一个或多个表的SQL查询来填充。然后，客户就可以用面向对象的方式来访问数据了。只有用SqlDataAdapter显式更新它们时改动才会反映到原来的数据库。同原来的数据源的变化保持一致并不是自动保证的。

Enterprise JavaBeans (EJB) [Sun04b]。EJB的Entity Bean在中间层（应用服务器）代表了数据库信息。这避免了昂贵的从数据库获取数据（资源获取）的操作。

对象缓存[Orac03] [Shif04]。对象缓存把这个模式应用于面向对象范型。在这种情形下，资

源是对象，它们的创建和初始化具有一定的开销。如果资源的使用允许缓存，那么对象缓存就可以避免这些昂贵的操作。

数据缓存[RoDe90] [Newp04]。数据缓存把这个模式应用于数据。数据被看做在某些情况下难以获得的资源。例如，这样的数据可能包含了复杂且昂贵的计算，或者有些信息可能需要从二级存储器中获取。这个模式使得已经获得的数据可以被重用，从而避免了昂贵的、在需要时重新获取数据的操作。

iMerge[Luce03]。iMerge EMS是iMerge VoIP（Voice over Internet Protocol） 硬件系统的元素管理系统，它使用了SNMP作为其通信接口。它使用Caching模式来优化网络元素之间连线的处理和可视化操作。

又见（See Also）

Pooling模式与Caching模式类似，但是具有很大的不同。Pooling模式背后主要的理念是重用没有标识的资源。它有助于避免重复获取和释放资源的开销，资源通常是没有任何标识的。Caching模式则与Pooling模式不同，因为Caching模式会在内存中维护资源的标识。这一不同使得Pooling模式可以透明地获取资源，而Caching模式则期望资源使用者获取资源。

Eager Acquisition模式通常使用Caching模式来管理预先获得的资源。

可以使用Evictor模式来清除缓存的数据。

Resource Lifecycle Manager模式可以在内部使用Caching模式来提供对有状态的资源的快速访问。

Cache Proxy模式[POSA1]可以用于隐藏缓存效果。TAO [Somm98][OCI04]用的Smart Proxies模式[HoWo03]会拦截远程调用，是特意为此设计的。

Cache Management模式[Gran98]专注于如何把缓存同Manager模式[Somm98]结合，Manager模式把对象的访问、创建和销毁集中化了。其描述更针对Java中的对象和数据库连接。

Page Cache模式[TMQH+03]是一个特化的缓存，用来改善访问动态创建的Web页面的响应时间。页面缓存同Web服务器相关联，Web服务器用页面缓存来存储访问过的页面，以它们的URL为索引。当相同的URL被请求时，Web服务器就查询缓存，并返回被缓存的页面，而不是再次动态生成页面的内容。

致谢（Credits）

感谢Ralph Cabrera和我们分享关于Caching模式的经验，并且提供了iMerge的已知应用。还要特别感谢Pascal Costanza，我们的EuroPLoP 2003审稿人，感谢他的杰出评价。我们还要感谢参加笔者研讨会的人们：Frank Buschmann、Kevlin Henney、Wolfgang Herzner、Klaus Marquart、Allan O'Callaghan和 Markus Völter.

3.2 Pooling模式

Pooling（池）模式描述了如何通过循环使用不再需要的资源来避免昂贵的获取和释放资源的操作。一旦资源被循环利用并置入池中，它们就失去了自己的标识和状态。

实例（Example）

考虑一个简单的基于Web的电子商务应用程序，它允许用户选择并订购一项或多项商品。假定解决方案是用Java写的，采用了3层构架。客户端是同Java servlet引擎（比如Tomcat）交互的Web浏览器。业务逻辑则实现于一个或者多个Java servlet，它们在Java servlet引擎中执行。这些servlet本身会使用Java Database Connection（JDBC）接口连接到数据库。图3-5展示了这样的情景，2个servlet在servlet引擎中执行，并连接到一个数据库。

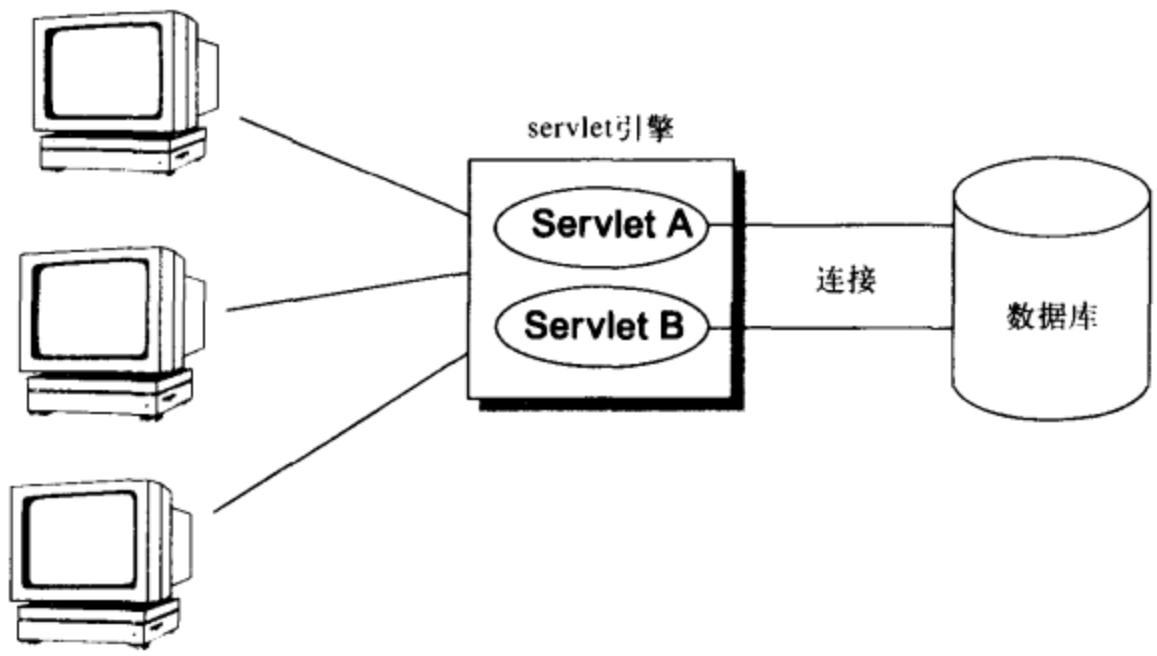


图 3-5

商品清单的大多数Web页面都是动态生成的，并且它们的内容取决于数据库。例如，为了获得清单中可获得的商品的列表以及它们的价格，一个servlet连接到数据库并执行查询。这个servlet使用查询的结果来产生HTML页面，这个页面会显示给用户。类似地，如果用户执行了一个购买操作，并且通过HTML表单输入了购买细节，那么这个servlet会连接到数据库并更新数据。

为了对数据库执行SQL语句，servlet需要获取到数据库的连接。一个平凡的实现是为每个servlet请求都建立一个到数据库的连接。这样的解决方案会非常没有效率，因为对于每一次用户请求它都会带来创建到数据库的连接的延迟。此外，这样的解决方案代价高昂，因为它可能会导致在短期内建立成千上万个到数据库的连接。

一种优化是把数据库的连接保存在servlet引擎的会话环境中。这使得我们可以在会话中重用连接。从属于同一次会话的用户发来的多个请求会使用同一个到数据库的连接。但是，一个在线清单可能会被上百个用户同步访问，所以，哪怕是这样的解决方案也会建立大量的数据库连接来满足需要。

环境（Context）

连续地获取和释放相同或类似类型的资源的系统，而且系统需满足高可伸缩性和高效率的要求。

问题（Problem）

很多系统都要求对资源可以进行快速并且可预测的访问。这样的资源包括网络连接、对象实例、线程以及内存。除了提供快速以及可预测的资源访问外，系统还要求解决方案对于所用资源的数目以及资源使用者的数目具有可伸缩性。此外，每一次用户请求应当在访问时间内经历很少的变化。所以，对资源A的获取时间不应当同资源B的获取时间有显著不同（A和B是相同类型的资源）。

为了解决上面提到的问题，需要解决这些作用力：

- 可伸缩性（Scalability）。被释放的资源应当被重用，以避免重复获取的开销。
- 性能（Performance）。应当避免反复获取和释放资源造成的CPU周期的浪费。
- 可预测性（Predictability）。资源使用者获取资源的时间应该是可预测的（即便从资源提供者直接获取资源的时间可能是不可预测的）。
- 简单性（Simplicity）。解决方案应该比较简单，以便尽量减少应用程序的复杂性。
- 稳定性（Stability）。反复获取和释放资源可能会增加系统不稳定性的风险。例如，反复获取和释放内存可能会对没有复杂的内存管理机制的操作系统导致内存碎片化。解决方案应该尽量减少系统的不稳定性。

Eager Acquisition和Lazy Acquisition之类的模式只解决了这些作用力的一部分。例如，Eager Acquisition模式使得资源获取具有了可预测性，而Lazy Acquisition模式则更关心如何避免不必要地获取资源。

简而言之，如何高效地获取和访问资源，同时还要保证没有牺牲可预测性和稳定性？

解决方案（Solution）

在池中管理一类资源的多个实例。这个资源池使得我们可以重用被资源使用者释放的不再需要的资源。被释放的资源会被放回到池中。

为了增加效率，资源池会在创建后预先获取固定数目的资源。如果对资源的需求超过了池中的资源数目，它会延迟地获取更多的资源。要释放不用的资源，可以有多种选择，比如Evictor模式或者Leasing模式所记录的方法。

当资源被释放并放回到池中时，资源使用者或者资源池应当让它失去其标识，这取决于所用的策略。之后，在被重用前，资源需要重新初始化。如果资源是一个对象，那么提供一个单独的初始化接口可能会很有用。资源使用者或者资源池不会用资源标识（在很多情况下是一个指针或者一个引用）来区分对象。

结构（Structure）

下列参与者形成了Pooling模式的结构：

- 资源使用者获取并使用资源。
- 资源是一个实体（比如内存或者线程）。
- 资源池（Resource Pool）管理资源，并响应资源获取请求，将资源交给资源使用者。
- 资源提供者（比如操作系统）拥有并管理资源。

下面的CRC卡片描述了参与者的职责以及它们的协作（见图3-6）。

Class Resource User	Collaborator
Responsibility • 获取并使用资源 • 把不用的资源释放到资源池中	• Resource • Resource Pool

Class Resource	Collaborator
Responsibility • 代表可重用的实体，比如内存或者线程 • 被从资源提供者处获得，并被资源使用者使用	

Class Resource Pool	Collaborator
Responsibility • 如果必要的话，事先获取资源 • 循环利用被资源使用者返回的资源	• Resource • Resource Provider

Class Resource Provider	Collaborator
Responsibility • 拥有并管理多个资源	• Resource

图 3-6

下面的类图展示了该模式的结构（见图3-7）。

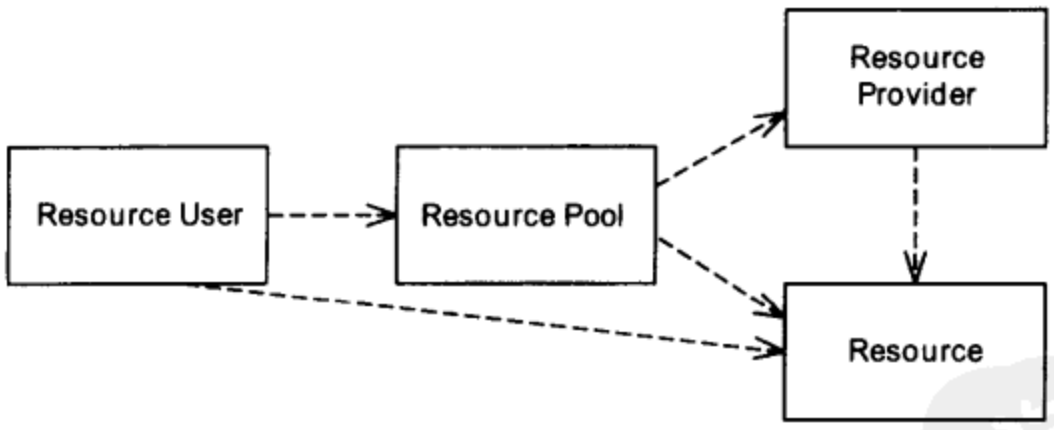


图 3-7

资源使用者只依赖于资源池以及实际的资源。资源由资源池从资源提供者获取。

动态（Dynamics）

下面的草图展示了参与者之间的交互（见图3-8）。

参与者之间的交互会略有不同，这取决于资源池是否一开始预先获取资源。

假定资源池一开始预先获取资源，那么来自资源使用者的后续的资源获取请求就会通过这些资源来满足。资源使用者不再需要资源时就将其归还给资源池。资源会被资源池循环利用，当有新的获取请求时资源池会再次提供它们。

下面的顺序图（见图3-9）描述了一个略微复杂一点的情形，在此情形下用户的资源获取请求会导致从资源提供者获取资源。资源会被按需获取，因为没有预先获取好的资源。资源使用者访

问并使用资源，当不再需要时，就把资源释放回资源池。资源池利用关于资源使用情况的统计数据来决定是否需要从池中清除资源。统计数据包含了使用特征，比如最近一次的访问以及访问频率。图3-9的第二部分展示了资源池基于统计数据决定清除被资源使用者释放的资源的情形。

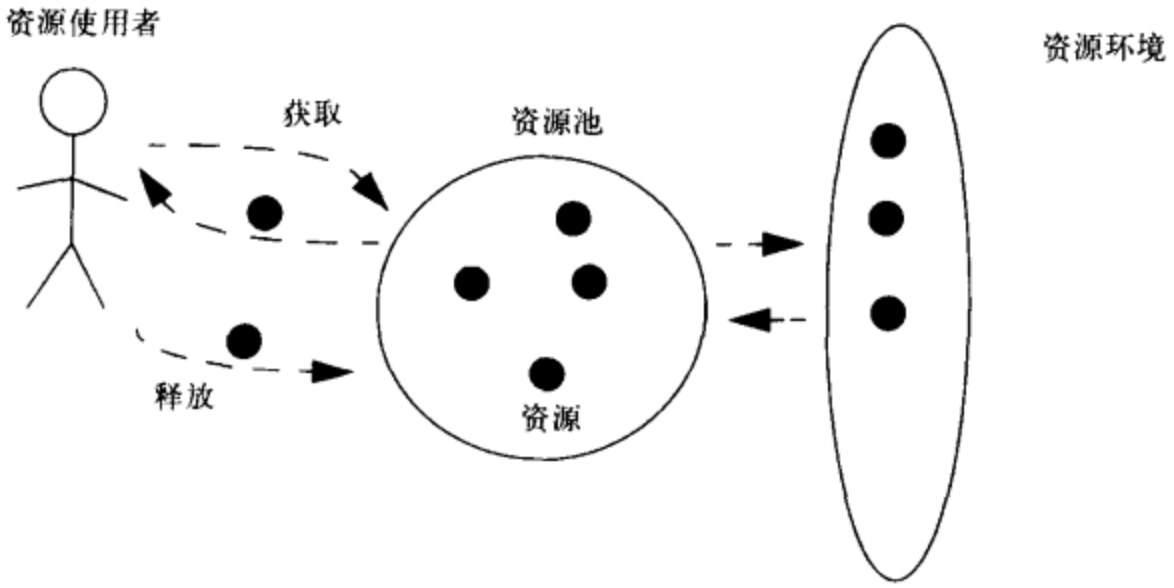


图 3-8

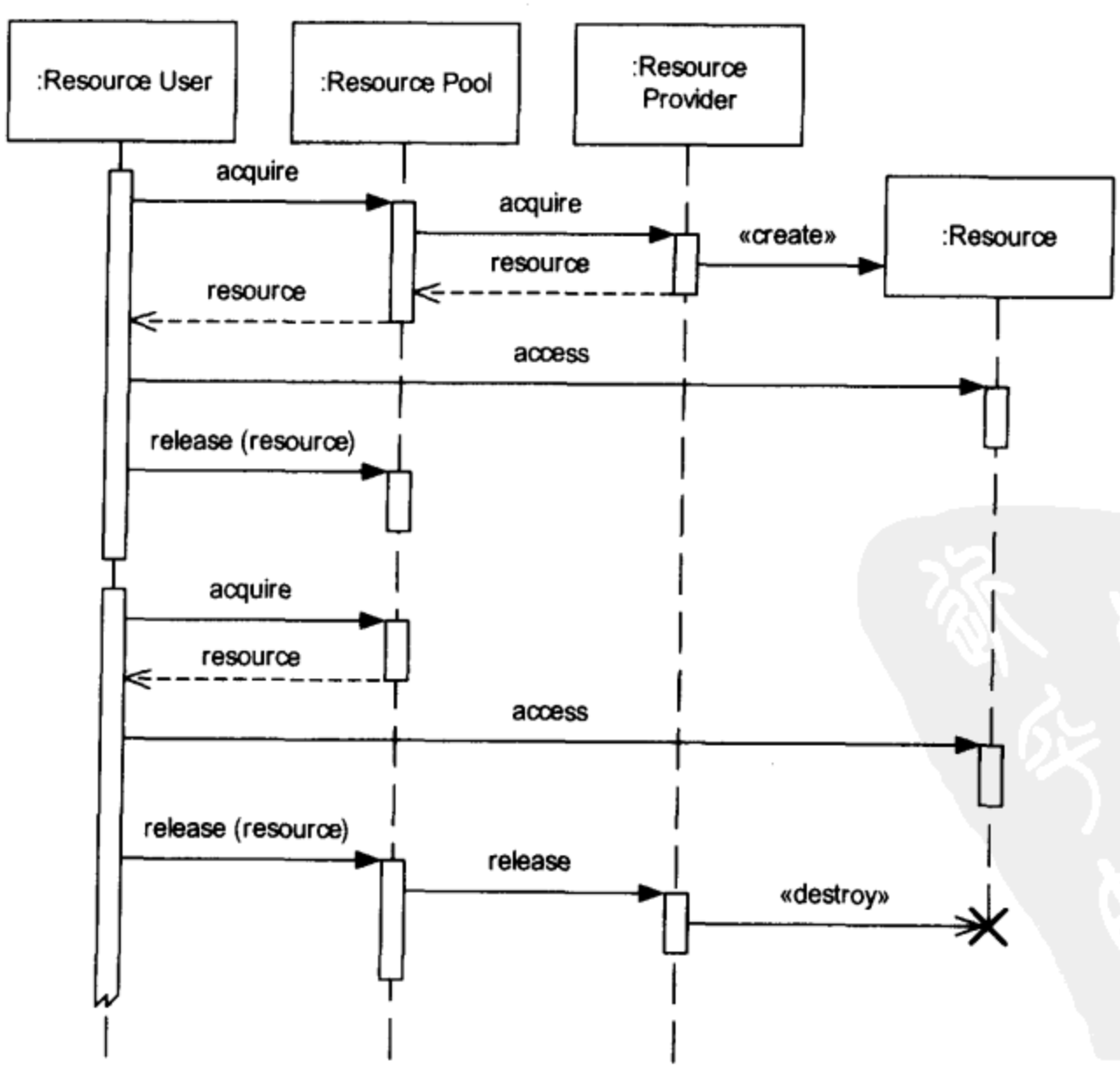


图 3-9

实现（Implementation）

实现Pooling模式包括了8步:

1）选择资源。找出会通过资源池获得好处的资源。通过根据类型把资源归入几个不同的资源池来简化资源管理。把不同类型的资源放入同一个池中会使得它们的管理复杂化，因为这会导致有必要划分多个子池以及进行专门的查找。

2）决定资源池的最大尺寸。为了避免资源耗尽，需要定义资源池维护的资源的最大数目。资源池中资源的最大数目等于预先获取的资源数目加上按需获得的资源数目。细节请参见Lazy Acquisition模式。资源的最大数目通常是在资源池的初始化时刻设置的，但是也可以根据可配置的参数（比如系统的当前负荷）来设置。

3）决定预先获取的资源数目。为了尽量减少运行时获取资源的时间，建议预先获取至少通常情况下会用到的资源的平均数目。这会减少应用程序常规执行时的资源获取。用户需求以及系统分析有助于判断要获取的资源的平均数目。重要的是，要记住太多的预先获取的资源反而会成为负担，因为它们会导致额外的资源竞争，所以应当避免。参见Eager Acquisition模式。

4）定义资源接口。提供所有放在资源池中的资源都需要实现的接口。这个接口有助于维护一组资源。

例如，用Java写的接口可能看起来像这样:

```
public interface Resource {}
```

Resource接口的实现维护了用来判断是否清除资源以及何时清除它的环境信息。环境信息包括时间戳和使用计数。

例如，在Java中Resource接口的实现（Connection类）可能看起来像这样:

```
public class Connection implements Resource
{
   public Connection () {
       // ....
   }
   // ....
   // Maintain context information to be used for eviction
   private Date lastUsage;
   private boolean currentlyUsed;
}
```

为了同遗留代码集成（遗留代码可能无法让资源实现Resource接口），可以引入Adaptor类[GoF95]。Adaptor类可以实现Resource接口并包装实际的资源。可以由Adaptor类来维护环境信息。

下面展示了用Java写的Connection类的Adapter类可能看起来是什么样子:

```
public class ConnectionAdapter implements Resource
{
   public ConnectionAdapter (Connection existingCon) {
       connection = existingCon;
   }

   public Connection getConnection () {
```

```
        return connection;
    }

    // Maintain context information to be used for eviction
    private Date lastUsage;
    private boolean currentlyUsed;
    private Connection connection;
}
```

5) 定义资源池接口。提供一个接口，让资源使用者可以获取和释放资源。

例如，用Java写的接口可能看起来像这样:

```
public interface Pool
{
  Resource acquire ();
  void release (Resource resource);
}
```

对Pool接口的实现可以维护一组Resource对象，当资源使用者试图获取资源时，可以从这组资源中获得。类似地，当资源使用者释放资源时，也会返回到这一组资源。

在Java中管理一组Connection对象的Pool接口的实现可能看起来像这样:

```
public class ConnectionPool implements Pool
{
    public Resource acquire () {
        Connection connection = findFreeConnection ();
        if (connection == null) {
            connection = new Connection ();
            connectionPool.add (connection);
        }
        return connection;
    }

    public void release (Resource resource) {
        if (resource instanceof Connection)
            recycleOrEvictConnection ((Connection) resource);
    }

    // private helper methods

    private Connection findFreeConnection () {
        // ...
    }

    private recycleOrEvictConnection (Connection con) {
        // ...
    }

    // ...
    // Maintain a collection of Connection objects
    private java.util.Vector connectionPool;
}
```

上面的代码展示了一种创建、初始化和清除连接的方式。当然，可以让这些操作变得更灵

活，比如使用Strategy模式[GoF95]和Abstract Factory模式[GoF95]。

下面的类图展示了前面提到的类的结构（见图3-10）。

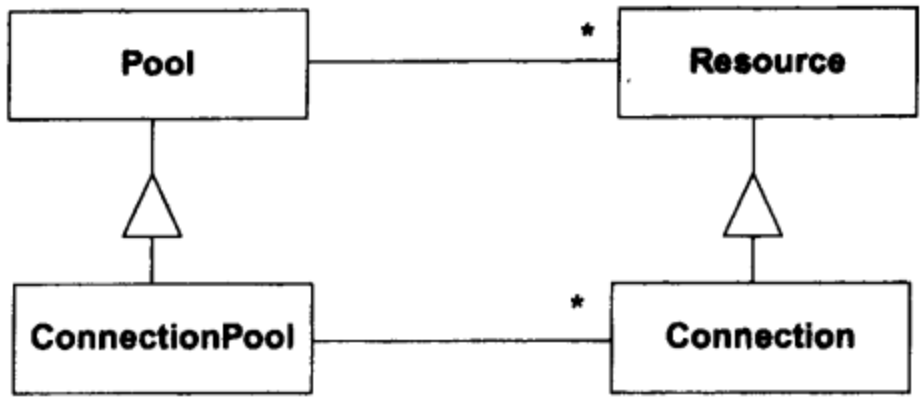


图 3-10

6) 提供资源清除。池中有很多资源意味着存在大量未被使用的资源，浪费了空间，降低了性能。为了尽量减少系统性能的下降，资源池的尺寸应当减少到一个合理的大小，这可以通过从池中释放一些资源来做到。

7) 决定资源循环利用语义。资源循环利用因资源类型不同而不同。例如，循环利用线程资源需要清楚它的堆栈以及初始化内存。对于循环利用对象的情况，Philip Bishop和Nigel Warren在它们的Java Tip[BiWa04a]中提供了一种把对象分解成多个更小的对象，然后循环利用这些对象的方式。

8) 决定失败处理策略。对于任何可以恢复的资源获取或者资源释放的失败，都应当处理异常以及/或者错误消息。如果不可能从失败中恢复，那么就应当抛出异常，或者向资源使用者返回空的资源（比如对于malloc()是返回NULL指针）。在循环利用失败的情况下，有别的模式可以帮助减少失败的影响。例如，前面提到的Java Tip[BiWa04a]也描述了如何通过把复杂的对象分割成更小的对象来循环利用损坏的对象。这些更小的对象或者被循环利用，或者被重新创建（如果损坏的话），然后它们被组装成复杂的对象。

实例解析（Example Resolved）

对于基于Web的商店的例子，数据库连接存放于连接池中。每次用户请求时，servlet都从连接池中获取连接，然后使用这个连接来访问数据库，查询或者更新。在访问过后，连接被释放到连接池中，此时的环境还是当前的请求。因为同步请求的数目小于当前用户的数目，所以总共需要的连接数目更少，昂贵的获取开销则被避免了。

图3-11表明了单个servlet如何从连接池（Connection Pool）中获得连接，以连接到数据库。如果连接池中没有可用的连接，而最大连接数目还没有达到，那么新的连接会被按需创建。

特化（Specializations）

下面的特化反映了这一模式在特定领域（比如内存分配、连接管理或者对象技术）的应用。现有的文献记录了一些特化。

Connection Pooling模式[HaBr01]。Connection Pooling模式在需要访问远程服务的可伸缩框架和应用程序中经常使用。这样的远程服务可以是数据库服务（比如通过JDBC），也可以是Web

Service（通过HTTP）。

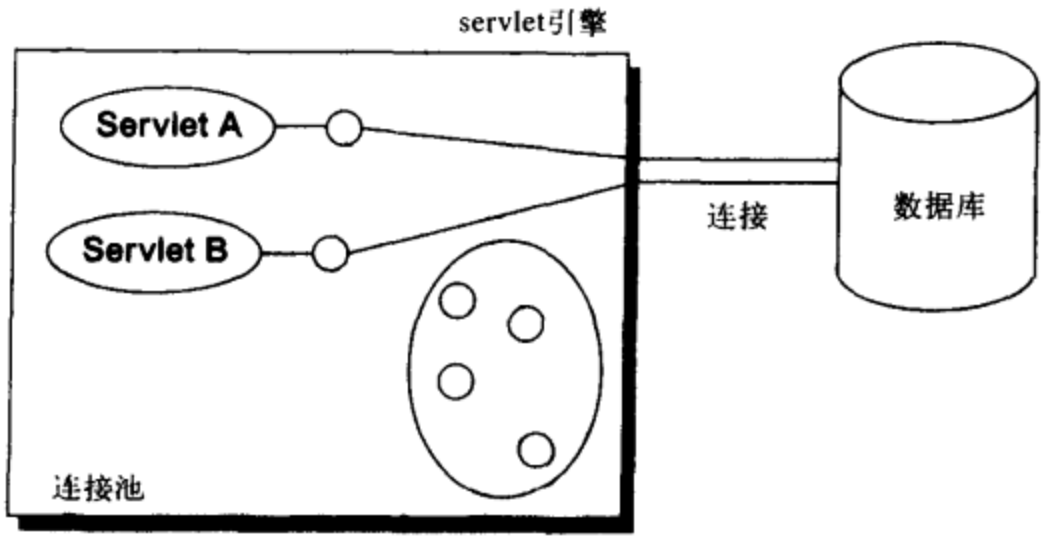


图 3-11

Thread Pooling模式[PeSo97]。Thread Pooling模式常用于本质上就是异步的环境。高度可伸缩的系统以及实时系统都把Thread Pooling模式用做它们管理多个线程的关键机制。使用Thread Pooling模式使得这些系统可以避免运行太多线程导致的资源耗尽。典型的线程池一开始具有有限数目的预先获取的线程，之后如果线程实例不够了再延迟地获取更多的线程。

Component Instance Pooling模式[VSW02]。Component Instance Pooling模式通常用于应用服务器。应用服务器通过预先实例化有限数目的常用组件来很快地为一开始的需求服务，从而优化了资源的使用。当这些组件不够用了，应用服务器就会随着额外的客户的请求实例化新的组件。如果客户停止使用某一组件，那么组件就返回到池中。

Pooled Allocation模式[NoWe00]。在这个模式中，当返回内存块时，预先分配好的内存池被循环利用。一般池分配器直到系统关机时才会释放内存。通常会安装多个池，每个池中的内存块具有不同的尺寸。这就避免了为了小的内存块请求而浪费大内存块。

Object Pool模式[Gran98]。Object Pool模式管理对特定类型（创建代价昂贵，或者只能创建有限数目的对象）的对象的重用。Object Pool模式和Pooling模式不同，因为Object Pool模式假定所有的资源都可以用一个对象来表示，而Pooling模式则没有这样的限制。Pooling模式的“实现（Implementation）”一节使用了对象，这仅仅是因为它们更易于传达关键的理念。在现实中，资源可以用对象来表示。但是，它们只是资源的代表，而不是实际上的资源。例如，不把连接这种资源表示成对象也能使用Pooling模式。

变体（Variants）

下面的变体描述了从Pooling模式引申而来的相关模式，这是通过扩展或者改变问题和解决方案而得到的。

Mixed Pool模式。不同类型的资源混合在一个池中，池的接口可能需要扩展，以便资源使用者可以区分其资源获取请求。所有的资源都由同一个策略来管理。接口允许获取每种类型的资源。

Sub-Pool模式。在某些情况下，把资源池进一步分为多个更小的池的做法是值得提倡的。

例如，对于线程池而言，你可以把它划分成多个子线程池，每个池都同特定的优先级范围相关联[OMG04b]。这就确保了当高优先级的任务需要执行时低优先级的任务获取线程不会导致资源枯竭。

结果（Consequences）

使用Pooling模式有几个**优点**：

- 性能（Performance）。Pooling模式可以改善应用程序的性能，因为它有助于减少花在费时的释放和重新获取资源的操作上的时间。在资源已经被资源池获取的情况下，资源使用者获取资源会变得很快。
- 可预测性（Predictability）。如果资源已经被预先获取，那么资源使用者执行平均数目的资源获取是在确定时间内完成的。使用合适的清除策略，大多数获取请求都可以通过资源池来满足。因为查找和释放以前获取过的资源是可预测的，所以资源使用者获取资源与总是从资源提供者获取资源相比，也较有可预测性。
- 简单性（Simplicity）。资源使用者不需要调用额外的内存管理例程。资源使用者或者透明地，或者直接从资源池获取和释放资源。
- 稳定性和可伸缩性（Stability and scalability）。如果需求超过了可用的资源，那么新的资源会被创建。因为要循环利用，资源池基于清除策略延迟了资源清除，这节省了代价高昂的对资源的释放和重新获取的操作，从而增加了系统的稳定性和可伸缩性。
- 共享（Sharing）。把池作为Mediator [GoF95]，这样未被使用的资源就可以在资源使用者之间共享。这对内存使用也有好处，并会降低应用程序的总体内存空间开销。

资源池没有同步对资源的访问。所以，如果需要同步，资源使用者必须自己做到。

使用Pooling模式也有一些**缺点**：

- 额外开销（Overhead）。对池中资源的管理会消耗额外的CPU周期。但是，这些CPU周期通常会少于释放和重新释放资源所需的CPU周期。
- 复杂性（Complexity）。资源使用者不得不显式地把资源释放回资源池。Leasing之类的模式可以用来解决这个问题。
- 同步（Synchronization）。在同步环境中，对资源池的获取请求必须被同步，以避免出现竞争条件和可能破坏关联的状态。

已知应用（Known Uses）

JDBC连接池[BEA02]。Java Database Connectivity（JDBC）连接是通过连接池管理的。JDBC规约第2版包含了引入连接池的标准机制。管理界面被称做javax.sql.DataSource，它为池中的连接对象提供了一个工厂。

COM+[Ewal01]。COM+服务的运行时（作为Windows操作系统的一部分）为COM+对象和.NET应用程序提供了对象池。.NET应用程序必须设置.NET的System.EnterpriseServices包中的配置属性。

EJB应用服务器[Iona04][IBM04a]。基于组件技术的应用服务器使用Component Instance Pooling模式来高效地管理组件实例的数目。

Web服务器[Apac02]。Web服务器必须为成千上百个同步请求提供服务。大多数请求都要快速响应，所以为每个请求创建新的线程是低效的。因此，大多数Web服务器使用Thread Pooling模式来高效地管理线程。在完成请求之后，线程会被重用。

又见（See Also）

Flyweight模式[GoF95]通过共享来高效地支持大量对象。Flyweight对象可以通过Pooling模式来维护。

这一模式又被称做Resource Pool模式[BiWa04a]。

Caching模式同Pooling模式相关。但是Caching模式是关于管理有标识的资源的。在Caching模式中，资源使用者会介意返回的是哪个被缓存的资源；而在Pooling模式中，资源使用者不介意资源的标识，因为池中的所有资源都是一样的。

致谢（Credits）

感谢西门子公司技术部的模式小组对这一模式的早期版本的反馈和评论。特别感谢我们的EuroPLoP 2002审稿人Uwe Zdun，以及参加笔者研讨会的Eduardo Fernandez、Titos Saridakis、Peter Sommerlad和Egon Wuchner的宝贵的反馈。

3.3 Coordinator模式

Coordinator（协调者）模式描述了如何通过协调涉及多个参与者（每个参与者都包含资源、资源使用者和资源提供者）的任务的完成来维护系统的一致性。这个模式提出了一个解决方案，使得在涉及多个参与者的任务中，或者所有参与者的任务都完成，或者一项任务都没有完成。这确保了系统总是处于一致的状态。

实例（Example）

考虑一个分布在多个处理节点构成的网络上的大规模系统（见图3-12）。分布在每个节点上的子系统包含了一个或多个服务，它们需要被频繁更新。假定把系统关闭然后更新所有的服务是不可行的，那么最常见的解决方案就是实现Deployer模式[Stal00]。使用Deployer模式，部署在每个子系统中的服务可以被动态更新，而不需要关闭系统。

为了让每个子系统都可以动态更新，每个节点都需要有一个软件下载组件。为了更新整个系统的配置，需要有一个集中监管组件来和每个节点上的软件下载组件通信。集中组件通常会向每个软件下载组件发送更新请求，然后下载组件会动态地更新部署在各个节点上的子系统。

考虑需要对系统配置执行整体更新的情形。集中监管组件需要向多个软件下载组件发送更新请求。假定一开始的几个软件下载组件成功地更新了它们各自的子系统，但有一个软件下载组件却未能更新其子系统，那么结果就是系统只有一部分被成功更新了。如果整体更新要求所有的子系统都成功更新，那么这样的结构就会导致系统处于不一致的状态。未能更新的子系统可能还在使用旧的版本不兼容的软件，使得整个系统都无法使用。

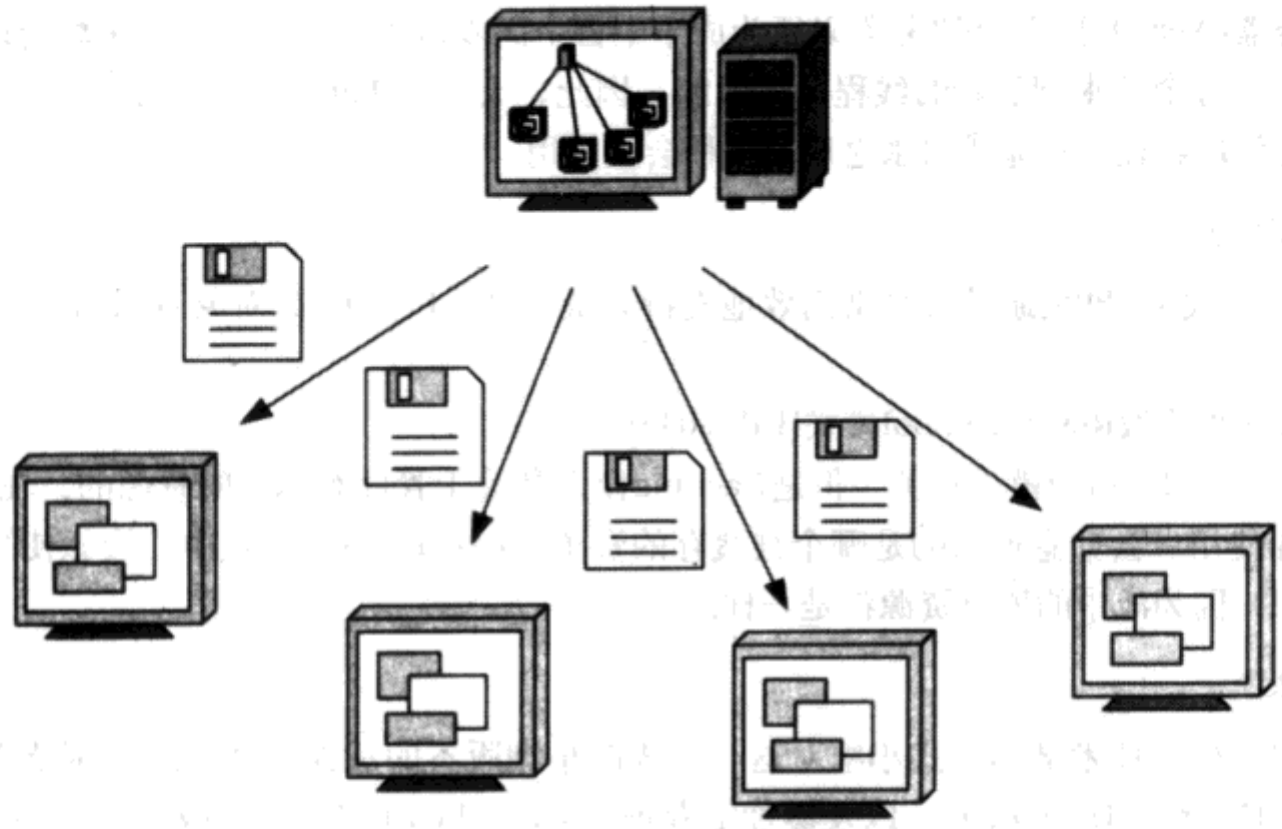


图 3-12

背景（Context）

协调执行同一个任务的多个参与者。任务被分配到各个参与者，每个参与者独立工作，以达成整个任务的成功。

问题（Problem）

很多系统都会执行涉及不止一个参与者的任务。一个参与者是一个主动实体，既包含资源使用者，也包含资源提供者。此外，在某些情况下，资源（比如服务）可以是主动的，所以会直接参与任务。参与者可能位于同一个进程中，也有可能跨越了多个进程、多个节点。每个参与者都会顺序执行任务的一部分。为了让任务获得整体的成功，每个参与者执行的工作都必须成功。如果任务成功执行，那么改变就应该让系统保持在一致的状态。但是，考虑一下如果一个参与者执行的工作失败了，会发生什么。如果这项工作位于任务执行序列的较后面的阶段，那么已经有很多参与者完成了其工作。这些参与者的工作会给系统造成改变。但是，失败的参与者无法对系统造成必要的改变，结果就是，系统的改变是不一致的。这可能会导致不稳定甚至出错的系统。在这样的系统中，部分失败甚至比全局失败更糟糕。

一个可能的解决方案是在参与者之间引入点对点的通信。每个参与者都把自己的工作结果传达给别的参与者，并采取必要的步骤以保证系统状态一致。但是，这样的解决方案要求所有参与者都关注任务涉及的其他所有参与者，这是不现实的。此外，这样的解决方案对参与者数量的可伸缩性也很糟糕。

为了解决这些问题，需要解决如下作用力：

- 一致性（Consistency）。一项任务应该要么为系统建立新的合法的状态，要么（若发生了错误）把所有的数据都恢复到任务开始前的状态。

- 原子性（Atomicity）。在涉及两个或多个参与者的任务中，或者所有参与者的工作都完成，或者所有工作都没有完成（虽然参与者是彼此独立的）。
- 位置透明性（Location transparency）。解决方案应当可以用于分布式系统，虽然分布式系统比整体式系统更有可能遭遇局部失败。
- 可伸缩性（Scalability）。解决方案应当对参与者的数目具有可伸缩性，而不会显著地降低性能。
- 透明性（Transparency）。解决方案对系统使用者应该是透明的，并且应该尽量减少对代码改动的要求。

解决方案（Solution）

引入协调者，负责所有参与者执行和完成任务。所有参与者执行的工作都分成两个阶段：准备（prepare）和提交（commit）。

在第一阶段，协调者要求每个参与者准备要完成的工作。每个参与者必须用这个阶段来检查一致性并判断执行结果是否会失败。如果某个参与者的准备阶段没有返回成功，那么协调者就停止任务的执行顺序。协调者会要求所有成功完成准备阶段的参与者都中止并恢复任务开始前的状态。因为参与者都还没有做出永久性的改变，所以系统状态保持了一致。

如果所有的参与者都成功地完成了准备阶段，那么协调者就发起每个参与者的提交阶段。参与者会在这个阶段做实际的工作。因为每个参与者都在准备阶段表明工作会成功，所以提交阶段会成功，整个任务被成功执行。

结构（Structure）

下列元素构成了Coordinator模式的结构：

- 任务（Task）是一个单元的工作，它涉及多个参与者。
- 参与者（Participant）是一个主动实体，它会完成任务的一部分工作。参与者可以包含资源使用者、资源提供者和资源。
- 协调者（Coordinator）是负责协调任务整体完成的实体。
- 客户（Client）是任务的发起者。客户指导协调者执行任务。

下面的CRC卡片描述了Coordinator模式的各元素的职责和它们间的协作（见图3-13）。

参与者之间的依赖关系在下面的类图中描述（见图3-14）。

动态（Dynamics）

Coordinator模式中有两个主要的交互。第1个是客户和协调者之间的交互，第2个是协调者和参与者之间的交互。

场景1：当客户要求协调者开始任务时，交互就开始了。之后，任务涉及的所有参与者都注册到协调者。然后客户指令协调者提交任务。此时，协调者先要求所有参与者执行准备阶段。如果有参与者表明准备阶段失败，那么协调者就中止整个任务。如果所有参与者的准备阶段都成功了，那么协调者就对所有参与者发起提交阶段。

Class Task	Collaborator
Responsibility • 工作单元	

Class Participant	Collaborator
Responsibility • 完任务的一部分工作 • 注册到Coordinator • 包含资源使用者、资源提供者和资源	• Task • Coordinator

Class Coordinator	Collaborator
Responsibility • 协调以确保一致地完成任务	• Participant

Class Client	Collaborator
Responsibility • 发起任务 • 提交任务	• Coordinator • Task

图 3-13

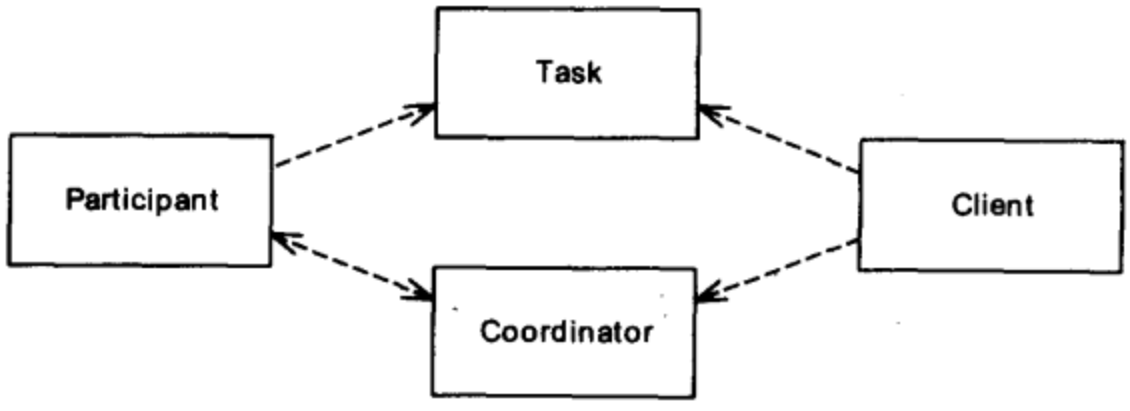


图 3-14

下面的顺序图展示了涉及两个参与者的场景，两个参与者的准备阶段都成功了（见图3-15）。

场景2：下面的顺序图展示了涉及两个参与者的场景，其中第二个参与者的准备阶段失败了。然后协调者就中止了任务，并要求第一个在准备阶段成功的参与者恢复它可能做过的改动（见图3-16）。

实现（Implementation）

实现Coordinator模式包括4步：

1）识别参与者。任何参与者，只要其工作需要被协调，那么就必须在一开始就识别出来。在资源管理的背景中，参与者可能是资源使用者、资源提供者或者资源本身。资源使用者可以成为参与者，如果它主动地试图获取、使用以及释放多个资源，从而需要协调的话；资源提供者也可以是参与者，如果它试图向资源使用者提供一个或多个资源的时候需要协调；资源（比如服务）也可以是参与者，如果它是主动的，并且因而能够直接参与任务。

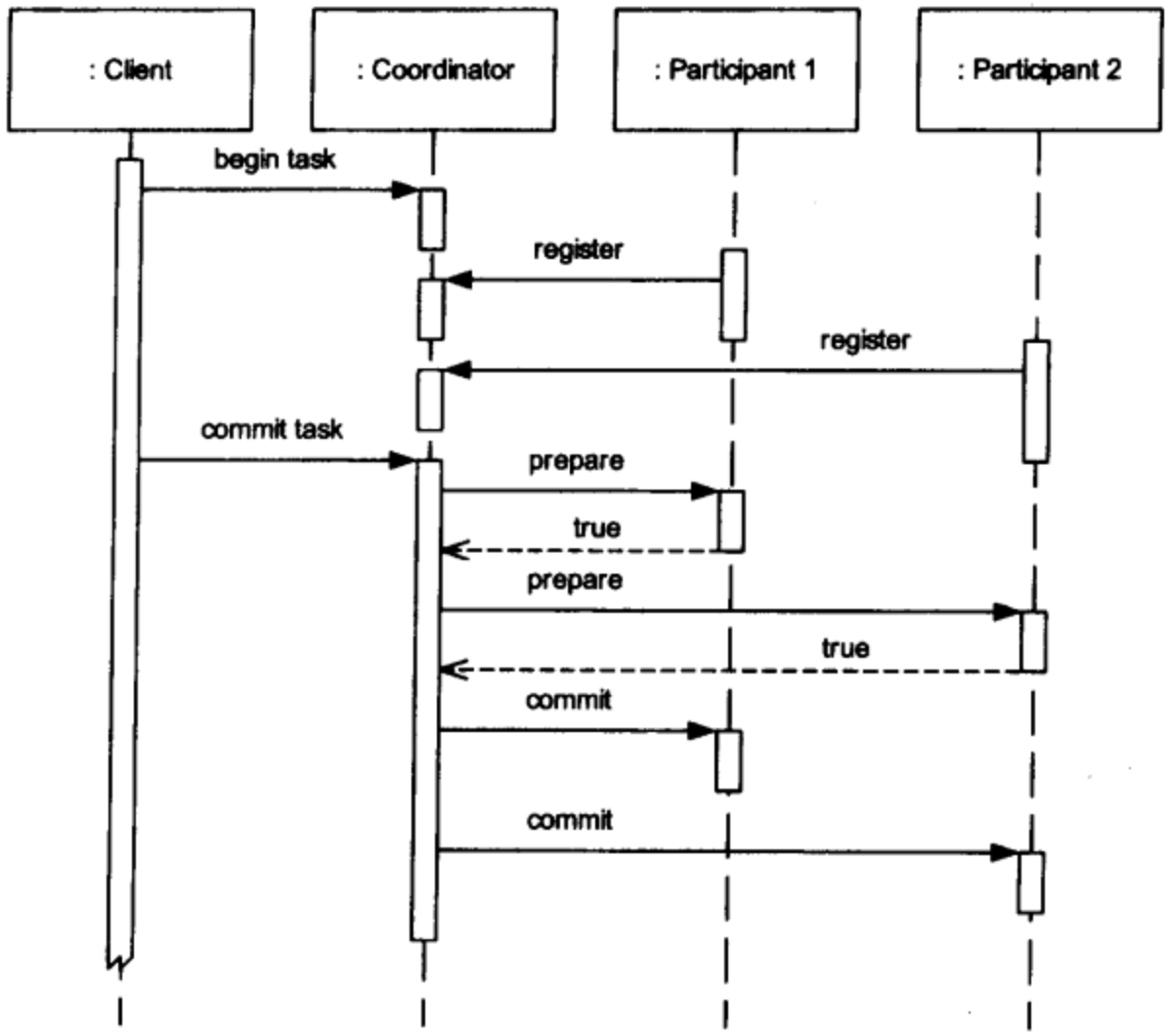

: Client
: Coordinator
: Participant 1
: Participant 2
begin task
register
register
commit task
prepare
true
prepare
true
commit
commit

图 3-15

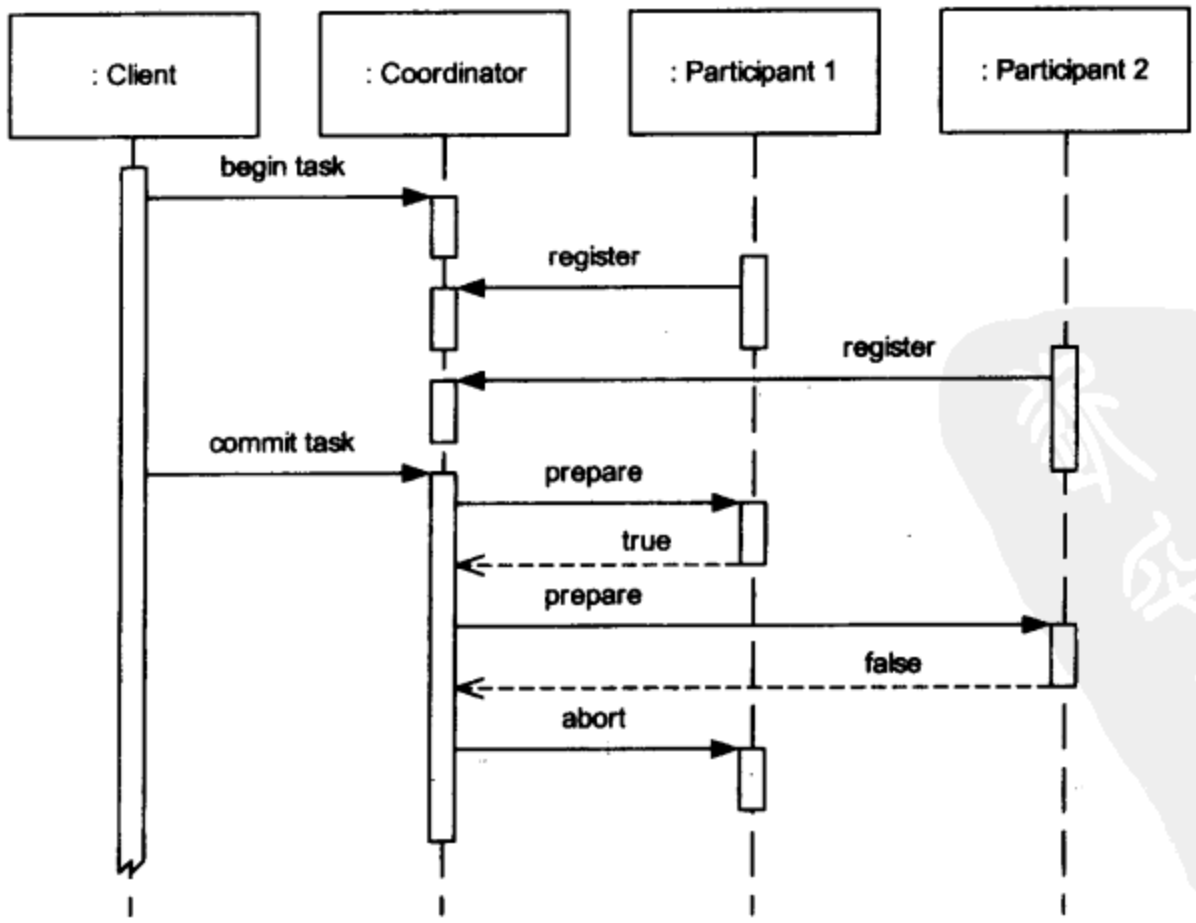

: Client
: Coordinator
: Participant 1
: Participant 2
begin task
register
register
commit task
prepare
true
prepare
false
abort

图 3-16

此外，任意其他使用或者管理资源的实体都可以看做是参与者，从而也需要被协调。例如，在Lazy Acquisition模式中，资源代理也可以看做是需要协调的参与者。资源代理可能需要获取用来代表多个资源的资源，因此资源获取任务可以被恰当地协调。类似地，在Eager Acquisition模式的情况中，提供者代理也可被看做是需要协调的参与者。

2) 定义协调接口。应该定义一个将被所有执行任务的一部分工作的参与者实现的协调接口。例如，用Java写的协调（Coordination）接口可能看起来像这样:

```
public interface Coordination {
   public boolean prepare ();
   public void abort ();
   public void commit ();
}
```

协调者使用prepare()方法来对每个参与者发起准备阶段。每个参与者都会使用这个阶段来检查一致性，并判断执行是否会导致失败。返回值true意味着准备阶段成功，因此提交阶段也会成功。返回值false意味着准备阶段失败，因此参与者将无法成功执行提交阶段。

如果在准备阶段任一参与者返回了false，那么协调者将会中止任务的执行。它会调用每个已经从准备阶段成功返回的参与者的abort()方法。这就向参与者表明，任务被中止，它们需要执行所有必要的清除操作。

3) 定义协调者接口。协调者接口应该为客户提供了开始或者结束任务的方法。此外，它还应该允许任务的参与者注册。在参与者可以注册它们之前，它们必须可以发现协调者。为了做到这一点，参与者可以使用Lookup模式。

例如，用Java写的协调者（Coordinator）接口可能看起来像这样:

```
public interface Coordinator {
   public void beginTask ();
   public void register (Coordination participant);
   public boolean commitTask ();
}
```

客户使用beginTask()方法来定义任务的开始。此时，协调者不做任何事情。当任务一开始，任务的参与者就用register()方法向协调者注册。一旦所有的参与者都注册了，客户就执行协调者的commitTask()方法。这对协调者而言意味着两件事情——首先，所有的任务参与者都已经注册；其次，协调者可以开始协调参与者执行任务了。现在协调者使用协调接口定义的两个阶段的提交协议来确保参与者完成任务。

4) 处理错误条件。使用协调者允许参与者表明它们负责的工作是否成功。如果任一参与者表明它的工作没有成功，那么协调者就可以中止任务，而不会导致系统状态的任何持久不一致性。如果所有的参与者都表明准备阶段成功，那么协调者就接着对所有参与者执行提交阶段。但是，还是会有这样的可能性：一个或者多个参与者在提交阶段失败，虽然它们的准备阶段成功完成了。这可能是由参与者无法控制的因素导致的，比如连接被断开了。为了处理这样的错误条件，参与者可以在执行它们负责的任务部分之前可选地维持状态信息。如果有一个参与者提交失败，那么协调者可以对剩下的提交成功的参与者调用rollback()方法。这会给参与者一次

机会，让它们恢复在执行任务之前的状态。不过，值得指出的是，这样的功能要求维护状态信息，可能代价高昂。更多细节请参见“变体（Variants）”一节的Three-phase Commit变体。

实例解析（Example Resolved）

考虑具有多个节点的分布式系统的例子。为了允许整个系统的动态更新，每个子系统都包含了一个下载组件。为了更新整个系统的配置，使用了一个集中监管组件来和每个节点的软件下载组件通信。

使用Coordinator模式，需要执行的任务就是更新整个系统。集中监管组件就是控制任务执行的客户。每个软件下载组件都扮演任务参与者的角色，并实现协调接口。参与者执行的子任务对应于对各个子系统的更新。

```
public class SwDownload implements Coordination {
  // Flag indicating current state
  private boolean downloading = false;

  public boolean prepare () {
      // Mark start of software downloading
      downloading = true;

      // Check if our state is consistent and
      // whether a software update can take place
      if (!this.validateState ())
          return false;

      // Now download the software but do not commit
      // the update. If download fails, return false.
      // ....
  }

  public void abort () {
      // Do not install the downloaded software.
      downloading = false;
  }

  public void commit () {
      // Commit/Install the downloaded software.
      // ...
      downloading = false;
  }
}
```

引入了一个协调者，集中监管组件将使用协调者来开始和提交任务。一旦任务开始，所有的软件下载组件都向协调者注册。这可以由软件下载组件自己来完成，也可以由集中监管组件来完成，参见“变体（Variants）”一节。当所有的参与者都注册之后，集中监管组件就要求协调者开始更新整个系统的任务。

```
public class CentralAdministrator {
  // Assume we have a reference to the coordinator
  Coordinator coordinator;

  public void updateSoftware () {
```

```
        // Ask the coordinator to begin the task
        coordinator.beginTask ();

        // Assume we have a list of sub-components stored
        // in an ArrayList. We now need to iterate over
        // this list and register all the sub-components
        // with the coordinator
        ArrayList components = getComponents ();
        for (Iterator i= components.iterator(); i.hasNext();) {
            Coordination swDownload = (Coordination) i.next();
            // Register the sub-component
            coordinator.register (swDownload);
        }
        // Now tell the coordinator to start the actual task
        coordinator.commitTask ();
    }
}
```

当协调者被要求提交任务时，它就会开始执行两步提交协议。协调者首先对每个软件下载组件发起准备阶段。在这个阶段，每个软件下载组件检查一致性，并判断执行更新会不会导致失败。然后，每个软件下载组件把软件更新下载下来，但不安装。如果在下载过程中出错，那么这个软件下载组件的准备阶段就会返回“失败”。

如果任一软件下载组件在准备阶段没有返回“成功”，那么协调者就中止任务的执行。然后，协调者要求所有成功执行了准备阶段的软件下载组件中止。任何子系统都没有发生改变，所以系统状态保持一致。

如果所有的软件下载组件都在准备阶段返回“成功”，那么协调者就发起提交阶段。在这个阶段，每个软件下载组件都安装下载的软件并更新其对应的子系统。因为每个下载组件都在准备阶段表明“更新将会成功”，所以提交阶段成功，进而导致整个系统更新成功。

```
public class SWCoordinator implements Coordinator {
    // Flag indicating if a task is in progress. Note that
    // this simple version only handles one task at a time.
    boolean taskInProgress = false;

    // List of registered participants
    private ArrayList list = new ArrayList ();

    public void register (Coordination participant) {
        if (!taskInProgress)
            return;
        if (!list.contains (participant))
            list.add (participant);
    }

    public void beginTask () {
        taskInProgress = true;
    }

    public boolean commitTask () {
        if (!taskInProgress)
            return false;
```

```
// Keep track of participants that complete prepare
// phase successfully
ArrayList prepareCompleted = new ArrayList ();

// Iterate over registered participants
// and use 2-phase commit
for (Iterator i= list.iterator() ;i.hasNext() ;) {
    Coordination participant = (Coordination) i.next();
    boolean success = participant.prepare ();

    if (success)
        prepareCompleted.add (participant);
    else {
        // Ask all participants that have completed
        // prepare phase successfully to abort.
        for (Iterator iter = prepareCompleted.iterator();
                iter.hasNext();) {
            Coordination p = (Coordination) iter.next();
            p.abort ();
        } // end for
        list.clear ();
        taskInProgress = false;
        return false;
    } // end else
} // end for

 // If we reach here, it means prepare phase of all
 // participants completed successfully. Now invoke
 // commit phase and assume it will succeed.
        for (Iterator i= list.iterator() ;i.hasNext() ;) {
            Coordination participant = (Coordination) i.next();
            participant.commit ();
        } // end for
        list.clear ();
        taskInProgress = false;
        return true;
    }
  }
```

值得注意的是，本节中的代码没有包含异常处理，这是为了简洁。如果在准备阶段抛出了异常，那么通常应该看做是失败，整个任务被中止。如果在提交阶段抛出了异常，那么对系统的更新应该回滚。参见“变体（Variants）”一节的Three-phase-Commit模式。

变体（Variants）

Third-Party Registration模式。参与者向协调者的注册操作不需要由参与者自身来完成。可以由第三方来完成，包含由客户完成。如果客户需要控制任务有哪些参与者，这个模式很有用。

Participant Adapter模式。任务的参与者不需要直接实现协调接口。这个对象可以被Adaptor[GoF95]所包含，由Adaptor来实现协调接口。协调者会调用Adaptor对象的prepare()方法，这个方法负责执行一致性检查并判断任务是否会成功。当协调者调用Adaptor对象的commit()方法时，它会把请求委托给实际的对象。这个变体使得整合遗留代码变得容易，不需要已经存在的类去实现协调接口。

Three-phase Commit模式。为了处理提交阶段可能失败，可以引入第三个阶段来补充两步提交协议。每个参与者都需要在执行它的任务部分之前维持状态信息。如果一个参与者的提交失败，那么协调者会对所有已经提交的参与者执行rollback()方法来执行第三阶段。这就给了参与者恢复到执行任务之前的状态的机会。如果所有参与者的提交阶段都执行了，那么协调者也会执行第三阶段，不过针对所有参与者调用的方法不同，是clear()。这个方法使得参与者可以丢弃为了预防提交阶段的失败而维护的状态。

结果（Consequences）

使用Coordinator模式有一些**优点**：

- 原子性（Atomicity）。Coordinator模式确保了在涉及两个或者多个参与者的任务中，或者所有参与者的工作都完成，或者所有工作都没有完成。准备阶段确保了所有参与者都能够完成工作。如果任一参与者在准备阶段返回失败，那么任务就不会被执行，这确保了没有任何参与者完成工作。
- **一致性**（Consistency）。Coordinator模式确保了系统状态保持一致。如果一项任务被成功执行，那么就为系统建立了新的合法的状态。另一方面，如果发生了任何失败，那么Coordinator模式确保所有的数据都回复到任务开始之前的状态。因为每个任务都是原子的，要么所有参与者都完成它们的工作，要么没有参与者完成任何工作。在这两种情况下，结果都是系统处于一致的状态。
- 可伸缩性（Scalability）。解决方案对于参与者的数据具有可伸缩性。增加参与者的数目不会影响任务的执行。随着参与者数目的增加，其中一个参与者失败的可能性也增加了。但是，使用这个模式，失败会在准备阶段被检测到，从而确保了系统处于一致的状态。
- 透明性（Transparency）。使用Coordinator模式对于用户是透明的，因为任务的两步提交式执行对于用户是不可见的。

使用Coordinator模式有一些**缺点**：

- 额外开销（Overhead）。Coordinator模式要求每项任务都分割成两个阶段，这会导致涉及所有参与者的执行序列被重复两次。如果参与者是分布的，那么这就意味着两倍的远程调用数据，这还可能会导致失去透明性。
- 额外的职责（Additional responsibility）。Coordinator模式为参与者增加了额外的职责，它们还需要向协调者注册。如果参与者是分布的，那么注册过程会导致参与者执行远程调用。

已知应用（Known Uses）

Java认证和授权服务（Java Authentication and Authorization Service，JAAS）[Sun04e]。JAAS的Login Context实现了Coordinator模式。JAAS支持可动态配置的层级登录模块执行认证的概念。为了确保要么所有的登录模块都成功，要么一个都没成功，Login Context分两个阶段执行认证步骤。第一个阶段（“登录”阶段），Login Context会调用配置好的登录模块，并要求每个模块只执行认证过程。如果所有必需的登录模块都顺利通过了这一阶段，那么Login Context会进入第二阶段，再次调用配置好的登录模块，要求每个模块正式地“提交”认证过程。在这

个阶段，每个登录模块都将其同相关的认证原则和证书关联。如果第一个阶段或者第二个阶段失败了，那么Login Context会调用配置好的登录模块并要求它们都中止整个认证尝试。每个登录模块都会清除其同认证尝试相关联的相关状态。

事务服务。很多事务模型和软件都实现了Coordinator模式。Object Transaction Service（OTS）[OMG04e]是一个分布式事务处理服务，是由Object Management Group（OMG）制定的。该规约提供的一个主要接口就是协调者接口，负责协调分布式事务。有很多OTS规约的实现，包括Inprise的ITS、Iona的Orbix 6 [Iona04]和Arjuna Solutions 公司的ArjunaTS。

Java Transaction Service（JTS）[Sun04d]实现了OMG OTS规约的Java对应物。JTS包含Transaction Manager，扮演协调者的角色。JTS Transaction Manager负责协调数据库事务。这些事务的参与者实现了事务保护的资源（比如关系数据库），并被称做“资源管理器”。

作为微软COM+服务一部分的Distributed Transaction Coordinator（DTC）[Ewal01]是一个事务服务器，实现了Coordinator模式。

数据库[GaRe93]。数据库广泛使用Coordinator模式以及两步提交的协议。如果数据库更新操作涉及几个相互依赖的表，而当数据库更新操作只完成了一部分时计算机系统出错了，那么使用两步提交可以确保数据库不会进入不一致或者不可操作的状态。

软件安装[Zero04b][FIK04]。大多数软件安装程序都实现了Coordinator模式。在准备阶段，会执行检查以确保安装任务会顺利执行。这包含了检查必要的磁盘空间和内存空间。如果准备阶段成功了，那么提交阶段就会执行软件的安装。这对于分布式软件安装尤其重要。

现实世界的例子1：Coordinator模式的一个现实世界例子是某些国家神父主持的婚礼。神父是协调者，新娘和新郎是参与者。在准备阶段，神父先问新娘和新郎：“你愿意和这个人结为伴侣吗？”只有双方都回答“我愿意”后神父才会宣布他们结为伴侣。

现实世界的例子2：另一个现实世界中的例子是中介者管理的双方交易。中介办理协调者的角色。在准备阶段，中介确保一方打算卖，另一方准备好了钱打算买。只有双方都满意时中介才会发起提交阶段，在这个阶段一手交钱一手交货。

又见（See Also）

Command Processor模式[POSA1]描述了记住操作序列并撤销操作的方法。它可以用做包装参与者执行的子任务的方法。

Master-Slave模式[POSA1]为工作和子任务的划分以及它们的协作建模。Master组件会把工作分发给Slave组件，并从Slave返回的结果计算出最终的结果。Master-Slave模式和Coordinator模式不同，因为它使用“分而治之”的策略，然后是结果的整合，而Coordinator模式则是关于每个子任务之后达到的状态的一致性。可以用Coordinator模式来扩展Master-Slave模式，以在把子任务的结果整合前确保一致性。

致谢（Credits）

感谢西门子公司技术部的模式小组，感谢我们的EuroPLoP 2002审稿人Kevlin Henney，以及参加笔者研讨会的Eduardo Fernandez、Titos Saridakis、Peter Sommerlad和Egon Wuchner的宝贵反馈。

3.4 Resource Lifecycle Manager模式

Resource Lifecycle Manager（资源生命周期管理器）模式引入了一个单独的Resource Lifecycle Manager，从而把资源的生命周期管理同它们的使用解耦合了。

实例（Example）

考虑一个需要为成千上万个客户提供服务的分布式系统。在客户和服务器之间会建立成千上万的网络连接，如图3-17所示。

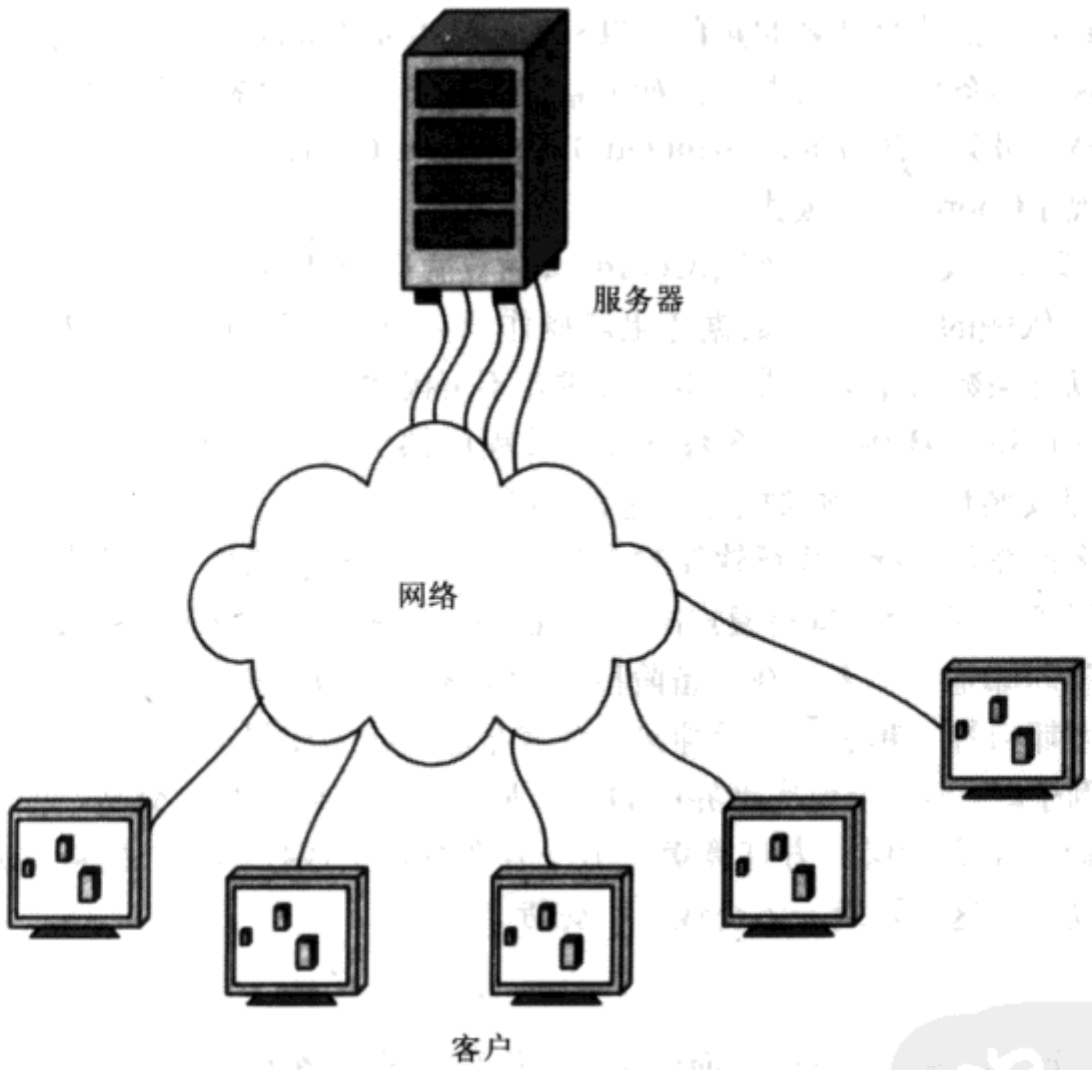


图 3-17

服务器通常向客户提供一个或多个服务，客户通过连接向服务器发送请求，对远程服务的同步调用的响应会作为结果通过同一个连接被送回。如果请求需要同响应解耦合，那么在客户和服务器之间就需要某种形式的异步通信方式，这样服务器就只需要打开到客户端的一个新连接，并使用这个连接来向客户端发起回调。例如，服务器可以使用回调来通知客户关于服务的事件。

对具有严格QoS需求的实时系统而言，维护几千个网络连接的任务就变得更加复杂了。结果就是，服务器的连接策略通常相当复杂。连接可能还与连接的请求优先级之类的属性相关联。

因为连接依赖于多个低层资源，所以不用时需要释放，以便确保系统的稳定性。结果就是，应用程序必须管理每个连接的复杂的生命周期场景。管理连接的生命周期的复杂性可能会影响

到应用程序的核心功能，结果就是使得业务逻辑难于理解和维护，因为它和连接管理代码纠缠在一起。

背景（Context）

大而复杂的系统，它们需要简化对其资源的生命周期管理。

问题（Problem）

建立大规模的系统很有挑战性。使大规模系统健壮并且具有可伸缩性更具挑战性。使大规模系统健壮并且具有可伸缩性的最重要因素是如何管理资源。系统中的资源可能具有很多种不同的类型，比如网络连接、线程、同步原语、服务等。网络连接代表了客户应用程序和分布式应用服务之间的通信渠道。高效地管理它们需要有能力判断何时建立连接、何时释放连接。线程对大规模系统特别重要，因为它们提供了应用程序不同部分之间的异步行为，例如可以把UI交互和典型的客户端功能以及服务提供解耦合。但是，高效地管理线程会很有挑战性，因为这涉及紧密监测它们的执行，并且判断何时创建新线程、何时释放不再需要的线程的能力。类似地，我们通常需要锁和令牌之类的同步原语来对应用程序的异步部分进行同步，并且使得它们的内部协调和交互成为可能。但是何时以及如何创建这些同步原语很重要，而且实现起来也很有难度。

要解决这些问题，有效且高效地管理资源生命周期，需要解决这些作用力：

- 可用性（Availability）。可用资源的数目通常和系统的整体尺寸不会同步增长。因此，在大系统中，有效且高效地管理资源很重要，这样才能确保在用户需要时有资源可用。
- 可伸缩性（Scalability）。随着系统变得更大，需要管理的资源数目也会增加，使得用户更难直接管理。
- 复杂性（Complexity）。大系统通常在资源之间会有复杂的相互依赖关系，很难跟踪这些关系。为了让资源可以在不再需要的时候正确而及时地释放掉，维护和跟踪这些相互依赖关系是很重要的。
- 性能（Performance）。通常很多优化的目的都是确保系统不会遇到任何性能瓶颈。但是，如果由单个资源使用者来进行这样的优化，可能会相当复杂。
- 稳定性（Stability）。如果资源使用者不得不管理资源生命周期的事项，它们可能会忘记释放资源，久而久之就会导致系统稳定性的问题。此外，还应该可以控制资源的获取，以确保在系统层次上可用资源不会发生“饥荒”，从而避免不稳定性。
- 相互依赖性（Interdependencies）。在复杂的系统中，相同或者不同类型的资源可能会相互依赖，这意味着资源的生命周期也会是相互依赖的，需要正确地管理。
- 灵活性（Flexibility）。对资源生命周期的管理应该是灵活的，可以支持不同的策略。策略应该提供钩子，从而允许配置资源管理行为。
- 透明性（Transparency）。资源生命周期的管理对资源使用者应该是透明的。特别是，资源使用者应该不需要被迫与资源管理的复杂性打交道。

解决方案（Solution）

把资源的使用同资源管理相分离。引入单独的Resource Lifecycle Manager（RLM），它的唯一职责就是管理和维护资源使用者用到的资源。

资源使用者可以用RLM来获取和访问特定的资源。如果被资源使用者请求的资源尚不存在，RLM会执行资源的创建工作。此外，RLM还允许用户请求显式地创建资源。

RLM知道当前的资源使用情况，所以可以拒绝来自资源使用者的资源获取请求。例如，当系统可用内存已经很少了，那么RLM可以拒绝资源使用者的分配内存请求。

RLM还控制它管理的资源的回收，或者对于使用者透明，或者响应使用者的显式请求。RLM基于考虑到可用计算资源（比如内存、连接和文件句柄）的合适的策略来维护资源。

RLM可以负责一类资源，也可以负责多种类型的资源。如果资源之间存在相互依赖关系，那么针对每一类资源的RLM会协同工作，这意味着它们需要维护资源间的依赖关系。这可以由一个中心RLM对其余各个RLM以及相互依赖的资源负责，也可以让一个单独的RLM只处理相互依赖关系，而把对相同类型的资源的管理留给各类资源的RLM。在层次分明的构架中，可以使用层级RLM，这样的RLM在每个抽象层次（比如OS、框架、应用程序层）都存在。

结构（Structure）

下面的参与者形成了Resource Lifecycle Manager模式的结构:

- 资源使用者获取和使用资源。
- 资源是一个实体（比如网络连接或者线程）。
- 资源生命周期管理器（Resource Lifecycle Manager）管理资源的生命周期，包括它们的创建/获取、重用和析构。
- 资源提供者（比如操作系统）拥有并管理资源。资源提供者本身也可以是一个在相同或者不同抽象层次上的资源生命周期管理器。

下面的CRC卡片描述了参与者的职责与协作（见图3-18）。

Class Resource User	Collaborator
Responsibility • 获取并使用资源 • 把不用的资源释放到生命周期管理器	• Resource • Resource Lifecycle Manager

Class Resource	Collaborator
Responsibility • 代表可重用实体，比如内存或者线程 • 被资源生命周期管理器从资源提供者获得	

Class Resource Lifecycle Manager	Collaborator
Responsibility • 协调资源的生命周期，包括创建/获取、重用和销毁	• Resource • Resource Provider

Class Resource Provider	Collaborator
Responsibility • 拥有并管理多个资源	• Resource

图 3-18

下面的类图描述了Resource Lifecycle Manager模式的结构（见图3-19）。

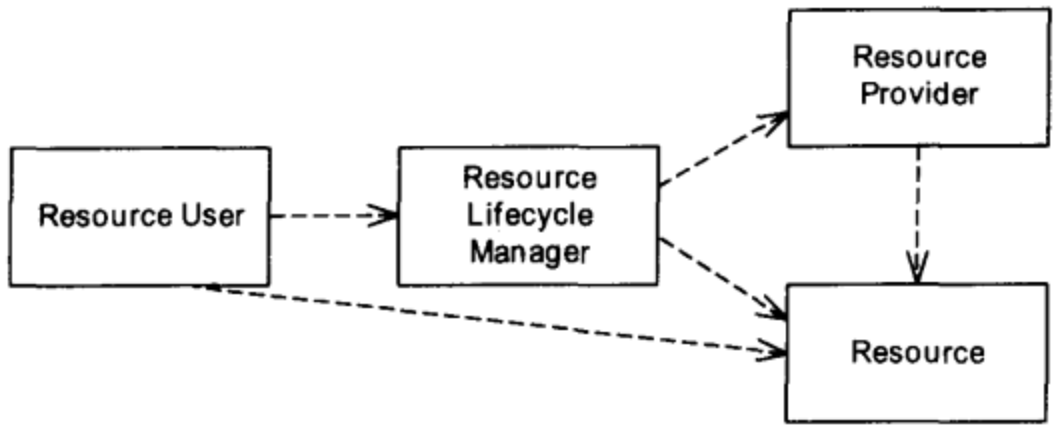


图 3-19

动态（Dynamics）

参与者间的交互可以用下面的图表来表示（见图3-20）。

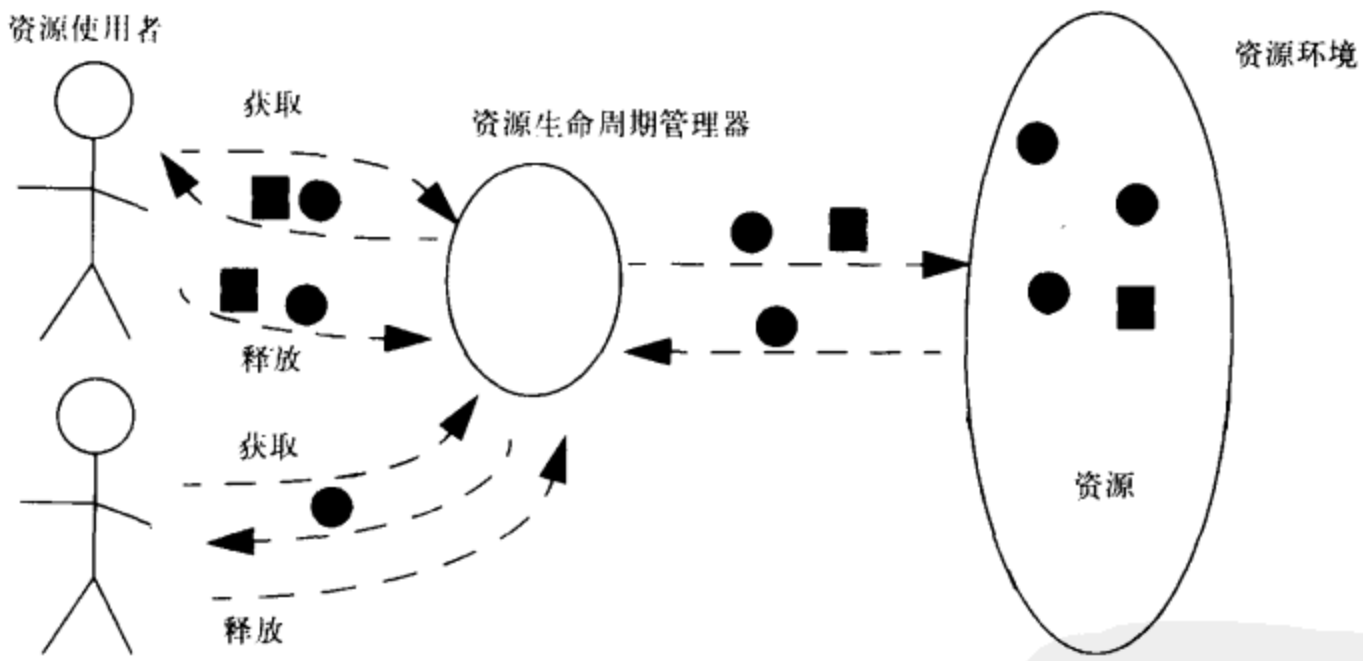


图 3-20

场景1：这个模式的动态行为包含如下活动：首先，系统启动并初始化RLM。资源使用者需要一个资源，所以试图从RLM获取资源。RLM接受了获取请求，并按照资源获取策略获得了资源，然后把资源传递给资源使用者。资源使用者访问并使用资源。

当资源使用者不再需要使用资源时，就会把资源还给RLM。RLM检查该资源到其他资源的依赖性，然后决定或者循环利用该资源，或者清除它。

下面的顺序图描绘了这些步骤（见图3-21）。

场景2：当资源使用者需要再次访问相同类型的资源时，资源使用者可以再次从RLM获取它。RLM可以运用Pooling模式和Caching模式作为优化，以避免从资源提供者获取资源的昂贵开销，参见下面的顺序图（见图3-22）。

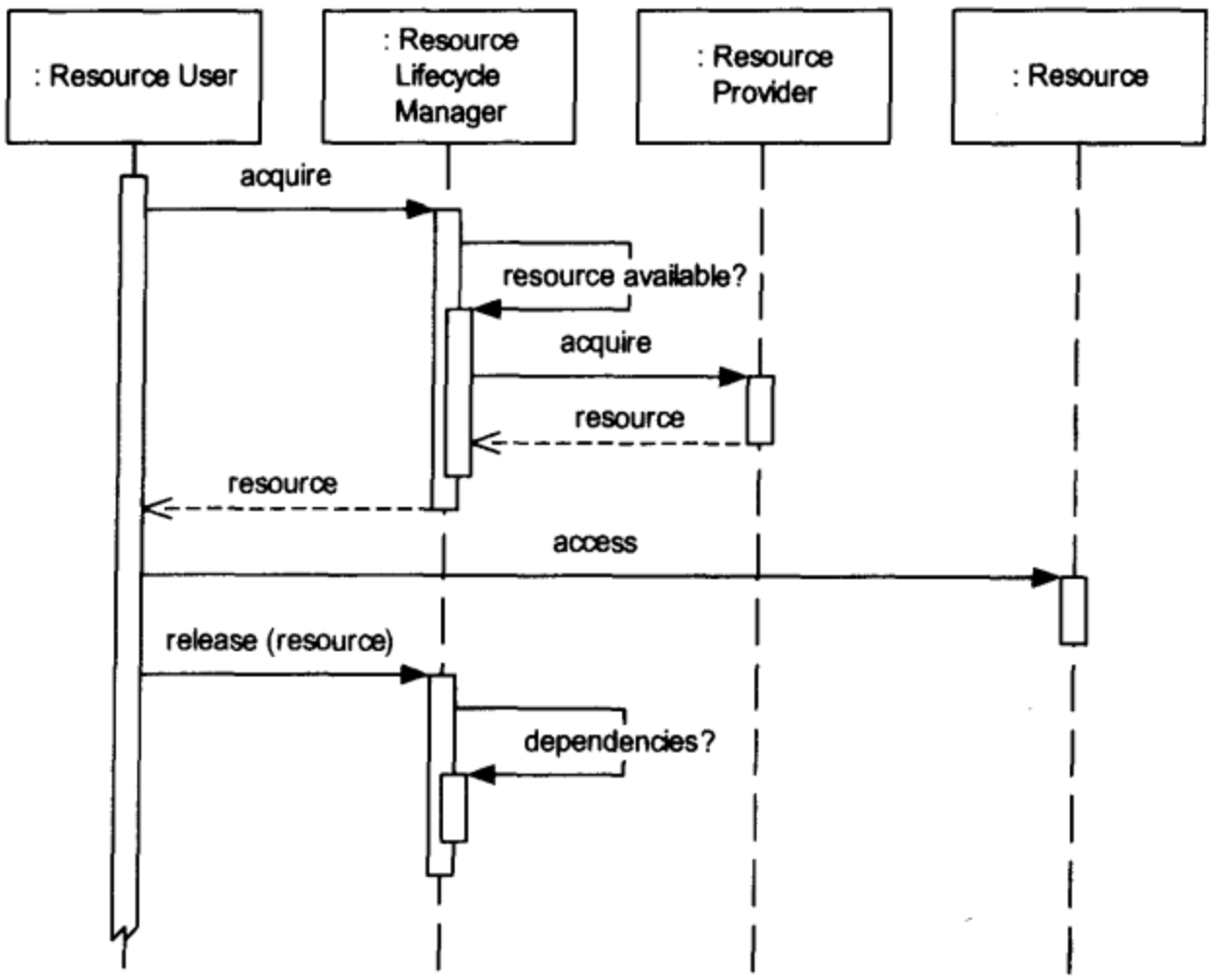

: Resource User
: Resource Lifecycle Manager
: Resource Provider
: Resource
acquire
resource available?
acquire
resource
resource
access
release (resource)
dependencies?

图 3-21

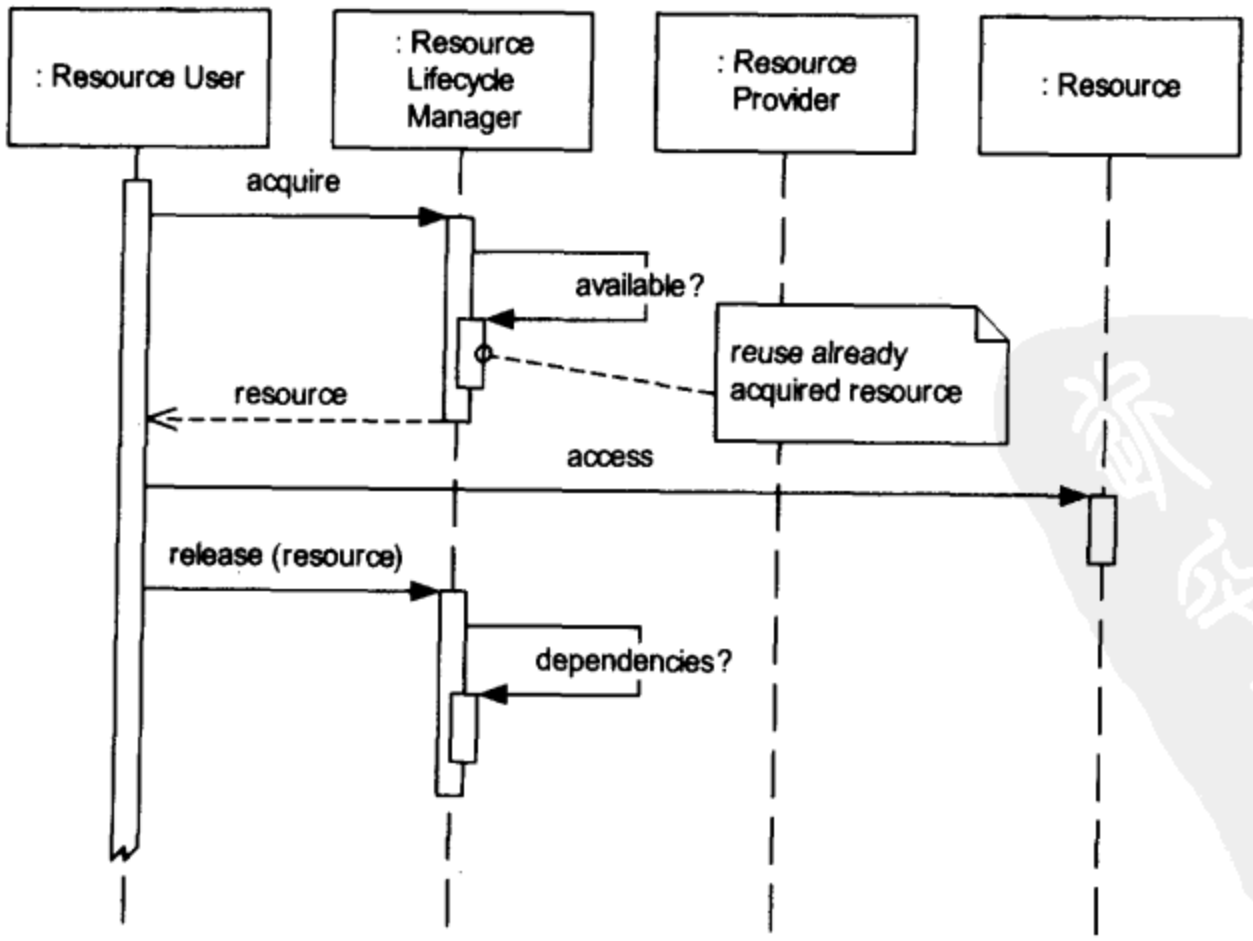

: Resource User
: Resource Lifecycle Manager
: Resource Provider
: Resource
acquire
available?
reuse already acquired resource
resource
access
release (resource)
dependencies?

图 3-22

实现（Implementation）

实现Resource Lifecycle Manager模式需要7步：

1）判定需要管理的资源。开发者首先需要识别出需要管理生命周期的所有资源。因为资源的类型多种多样，所以应用程序可以为不同类型的资源提供多个RLM，例如一个RLM用于处理进程、线程、文件句柄和连接之类的计算资源，另一个RLM用于维护应用程序组件。不过，也可以用一个RLM来处理不同类型的资源。当需要在不同类型的资源之间维护复杂的相互依赖关系时，这样的解决方案可能比较有效（参见实现步骤4）。如果只需要一个RLM实例，它应该被实现为Singleton[GoF95]。

2）定义资源创建和获取语义。开发者需要决定RLM如何创建或者获取资源。

这既包括决定资源何时创建或者获取，也包括资源如何创建或者获取。Eager Acquisition、Lazy Acquisition和Partial Acquisition之类的模式可以用来控制何时获取资源，而Factory Method模式[GoF95]和Abstract Factory模式[GoF95]则可以控制资源如何创建。请注意，通常资源由RLM来获取，所以RLM可以对这些资源的生命周期有完全的控制。但是，也可能存在这样的情况：一些资源不是由RLM获取或者创建的，但还是需要由RLM来管理。

为了确保系统的稳定性，RLM可以以多种理由（包括资源不够用的情况）拒绝资源使用者的资源获取请求。

Pooling模式的实现可以管理通常会在RLM初始化时预先创建好的资源（用Eager Acquisition或者Partial Acquisition模式创建）。Eager Acquisition模式在资源被访问之前完整地获得资源，所以在获取之后资源可以立即投入使用。但是，创建或者获取大的资源可能会花较长的时间。Partial Acquisition模式则有助于降低预先获取资源的时间，这是通过分步获取资源来做到的。若使用Lazy Acquisition模式，那么整个资源获取被延迟到实际访问资源的时候。

3）定义资源管理语义。RLM的一个主要职责是有效且高效地管理资源。频繁的资源获取和释放可能代价高昂，所以RLM通常会使用Caching和Pooling这样的模式来优化对资源的管理。Pooling模式可以用于保持一个固定数目的资源始终可用。这一策略对于管理线程和连接这样的资源特别有用，因为这些资源的使用者在重新获取时通常不依赖于它们的标识。而Caching则比较适用于必须保持标识的资源。

例如，Caching模式经常用于任务涉及的有状态的应用程序组件，因为它们必须保持自己的状态以及标识。一旦任务完成，那么就不再需要该组件了，直到相同的任务被再次执行。为了避免降低应用程序的服务质量，从内存中临时去除不用的组件可能会有帮助，这样它们占据的空间和计算资源就可以被使用中的组件所获得。Passivation模式[VSW02]描述了组件的“冬眠”和重新激活。把Caching和Passivation模式一起使用有助于限制总体资源消费。

4）处理资源的依赖关系。在很多应用程序中，资源是互相依赖的。因此，第一步是把不同资源的生命周期隔离到单独的RLM中去，以便简化管理。但是，这样一来就很难进行基于依赖资源的优化。

在上面的例子中，依赖资源可以包括通过特定的连接进行访问的应用程序服务。在实时环境中，应用程序服务的实现经常直接同来自客户的不同优先级的连接相关联，这样就可以从端到端来确保优先级。因此，清除这样的连接会影响到服务的行为，包括它使用的资源。如果服

务所依赖的连接被清除了，它可能会变得不可访问。因此，为了管理连接、服务和它们的依赖关系，可以考虑使用一个具有公共职责的RLM。

这样的RLM的确切可行性，以及它的实现，与应用程序背景和资源类型密切相关。一个可能的解决方案是把相互依赖的资源划入一个组。可以用Builder模式[GoF95]来控制多个依赖资源的创建。这样的分组可以用于控制相互依赖的资源的获取、访问和释放。例如，可以配置一个策略，确保若一个资源被释放，那么组中依赖于它的资源也被自动释放。

5) 定义资源释放语义。当资源不再被需要时，它们应当自动被RLM释放。Leasing和Evictor这样的模式可以用来控制何时以及如何释放资源。Evictor允许从缓存中受控地删除使用频率较低的资源。为了避免释放依然被引用的资源，RLM可以使用Leasing模式，让它指定资源可被资源使用者获得的时间。一旦过期，资源就可以安全地从RLM释放了。此外，也可以用Garbage Collector[JoLi96]来识别不用的资源（没有被任何资源使用者或者其他资源引用的资源）并清除它们。为了安全地清除资源，可以考虑使用Disposal Method模式[Henn03]。

6) 定义资源访问语义。被创建或获取的资源应该易于访问。RLM可以使用Lookup之类的模式来简单地访问资源。

7) 配置策略。对于前面的每一步，RLM都应该允许配置不同的策略来控制如何管理资源的生命周期。例如，如果资源很昂贵，那么就应该使用Lazy Acquisition模式尽可能晚地获取它，并且使用Evictor模式尽可能早地释放它。测量资源的代价是和应用程序相关的。例如，如果一个资源消费大量的内存，或者在资源使用者之间很抢手，那么可以认为它很昂贵。另一方面，如果资源不是很昂贵，并且使用频繁，那么应该用Early Acquisition模式尽早获取它，并且在应用程序的生命周期中都保留它。在动态环境中，可以用反射机制[POSA1]来根据环境适配配置策略。当必须管理资源之间的依赖关系时，可以用Coordinator模式来同步它们的释放。

实例解析（Example Resolved）

引入一个组件，负责管理资源的生命周期，从而让应用程序从这一职责中解脱出来（见图3-23）。

这解除了连接的管理与应用程序的业务逻辑之间的耦合。

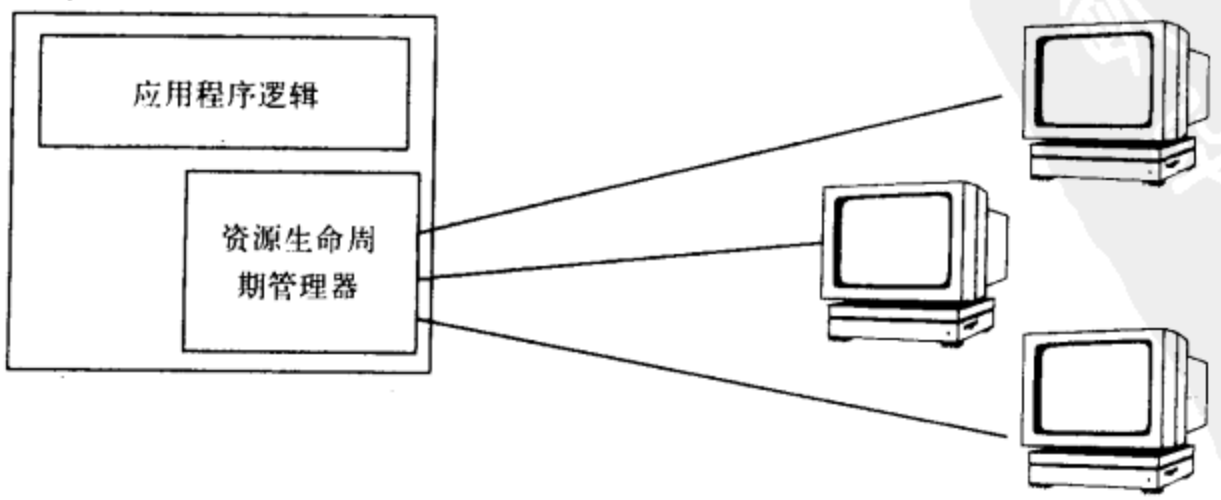


图 3-23

在分布式应用程序的客户端和服务端都要引入资源生命周期管理器。客户端会使用资源生命周期管理器来请求到服务端的新连接。一旦这些连接传递给客户，它们的生命周期就由资源生命周期管理器来管理了。

在下面的段落中，我们会展示一个资源生命周期管理器的具体实现。ResourceLifecycleManager类为Resource Lifecycle Manager模式提供了基本的接口。使用这个类可以获取和释放不同类型的资源。ResourceLifecycleManager使用了多个资源池来管理资源的生命周期。如果一个资源在相应的池中没有找到，那么会用工厂来创建资源。一旦资源被创建，用户必须将其放入一个资源组，以便让它可以被正确地管理。假定每个资源都至少属于一个组。请注意，为了简洁起见，我们省略了所有错误和异常处理代码。

```
public class ResourceLifecycleManager {

  public ResourceLifecycleManager () {
      // Create and set up all the appropriate pools and
      // factories
  }

  public Resource acquire (ResourceType type) {

      Pool pool = (Pool) resourcePools.get (type);
      if (pool != null) {
          Resource resource = pool.acquire ();
          if (resource != null)
              return resource;
      }
      // The Resource pool did not have a resource, so try
      // creating one using the appropriate factory.

      ResourceFactory factory =
          (ResourceFactory) resourceFactories.get (type);
      if (factory != null)
          return factory.create();

      // Handle error... no resource available
      return null;
  }

  public void release (Resource resource) {
      ResourceGroup group = this.findGroup (resource);
      if (group != null)
          group.release ();
  }

  // Set the factory for the type of resource
  public void setFactory (ResourceType type,
                          ResourceFactory factory) {
      resourceFactories.put (type, factory);
  }

  // ... (group related methods)
}
```

acquire()方法会试图获得给定类型的资源。这个方法首先找到给定类型所对应的资源池。然后它会试图从池中获取资源。如果在池中找不到资源，那么这个方法就会寻找这个资源类型所对应的工厂，并用工厂来创建资源。请注意，何时真正创建或者获取资源是由工厂配置的不同策略决定的。

release()方法会释放资源以及所有依赖资源。依赖资源是指属于该资源所属的ResourceGroup的资源。ResourceLifecycleManager内部使用Coordinator模式来确保要么所有的资源都被释放，要么一个资源都没有被释放。在ResourceLifecycleManager发起的两步协议中，每个资源都扮演参与者的角色，ResourceLifecycleManager则扮演协调者的角色。

对于所有和组相关的功能，ResourceLifecycleManager类提供了下面的一组方法。

```
public ResourceLifecycleManager {

   // ... (already described methods)

   public ResourceGroup createGroup (String groupID,
                                     Resource resources []) {
       ResourceGroup group = new ResourceGroup (groupID);
       for (int i = 0; i < resources.length; i++) {
           Resource resource = resources[i];
           group.add (resource, r.type ());
       }
       return group;
   }

       public void addResourceToGroup (Resource resource,
                                       ResourceGroup group) {
           group.add (resource, resource.type ());
       }

       public boolean release (ResourceGroup group) {
           // Iterate over all resources in the group and release
           // them according to the Coordinator pattern
           // ...
       }

       private ResourceGroup findGroup(Resource resource) {
           // Find the group to which this resource belongs
       }
   )
```

createGroup()方法会创建一个具有标识的组，然后把资源加到组中。addResourceToGroup()方法把增加资源的工作委托给ResourceGroup类的方法。每个被管理的资源都必须实现Resource接口。

```
public interface Resource {

   // Called before the resource is evicted
   public boolean beforeEviction ();

   // Return the type of the resource
   public ResourceType type ();
}
```

钩子方法[Pree94] beforeEviction()会在资源被清除前调用。资源应该首先判断它自己是否处于允许被清除的状态。如果不是，那么这个方法应该返回false，资源就不会被清除。否则的话，这个资源就应该在被清除之前执行必要的善后工作，并且应该返回true。Connection资源实现了Resource接口，如下所示:

```
public class Connection implements Resource {

   public void authenticate(SecurityToken token) {
       // ...
   }

   public boolean beforeEviction () {
       if (!consistentState)
           return false;
       // Release any other resources and clean up
       // ...
       return true;
   }

   public ResourceType type() {
       return ResourceType.CONNECTION;
   }

   private boolean consistentState;
}
```

钩子方法beforeEviction()首先检查资源是否可以被释放。为此它使用了布尔标志consistentState，用来标记资源是否处于可以被清除的状态。在我们的例子中，连接可以用安全令牌来验证。安全令牌也实现了Resource接口，它表示连接所依赖的次级资源。

使用验证连接并且希望把两种资源的生命周期都委托给RLM管理的应用程序会这样使用ResourceLifecycleManager:

```
ResourceLifecycleManager rlm =
   new ResourceLifecycleManager();

Connection connection = null;
SecurityToken token = null;

// Acquire the resources
connection =
   (Connection) rlm.acquire(ResourceType.CONNECTION);

token =
   (SecurityToken) rlm.acquire(ResourceType.SECURITY_TOKEN);

Resource resources [] = new Resource [2];
resources[0] = connection;
resources[1] = token;

// Put the resources in a group so they can be
// managed properly
ResourceGroup group =
   rlm.createGroup ("xyz123", resources);
```

```
// Use the resources ...
connection.authenticate (token);

// ...

// Release all the resources by releasing the group
rlm.release (group);
```

这段代码显式地提供了用于同组相关的资源的接口方法。如果应用程序不想承受组管理的额外负担，可以用Wrapper Facade模式[POSA2]来隐藏这一复杂性。

结果（Consequences）

使用这个模式有几个**优点**:

- 效率（Efficiency）。单个使用者来管理资源的效率比较低。Resource Lifecycle Manager模式允许对资源进行协调和集中式的生命周期管理，这使得进一步的应用程序优化成为可能，并且降低了总体的复杂性。
- 可伸缩性（Scalability）。使用Resource Lifecycle Manager模式允许更有效的资源管理，使得应用程序可以更好地利用可用的资源，这使得应用程序可以承受更高的负荷。
- 性能（Performance）。Resource Lifecycle Manager模式可以确保多个层次的优化成为可能，从而从系统获得最佳的性能。通过分析资源的使用情况和可用性，可以用不同的策略来优化系统性能。
- 透明性（Transparency）。Resource Lifecycle Manager模式使得资源管理对于用户透明。可以配置不同的策略来控制资源的创建和获取、管理以及释放。通过把资源使用和资源管理解耦合，RLM降低了使用复杂性，从而让资源使用者变得轻松。
- 稳定性（Stability）。Resource Lifecycle Manager模式可以确保只有在有足够数目的资源的情况下才会把资源分配给用户。这避免了在资源使用者可以直接从系统获取资源的情况下容易导致的资源“饥荒”，从而有助于提升系统的稳定性。
- 控制（Control）。Resource Lifecycle Manager模式允许更好地控制对相互依赖的资源的管理。通过维护和跟踪资源间的依赖关系，RLM可以正确并且及时地在资源不再被需要时释放它们。

使用这个模式有几个**缺点**:

- 单点失败（Single point of failure）。RLM中的bug或者错误会导致应用程序的很多部分失败。冗余概念只能起到部分作用，因为复杂性进一步增加，性能则受到进一步的限制。
- 灵活性（Flexibility）。当单个资源实例需要特殊对待时，Resource Lifecycle Manager模式可能灵活性很不够。

已知应用（Known Uses）

组件容器。容器管理应用程序组件的生命周期，并且提供独立于应用程序的服务。此外，它还管理组件使用的资源的生命周期（参见[VSW02]中的多种Container and Managed Resource模式）。很多技术都提供了容器功能，包括J2EE Enterprise JavaBeans（EJB）[Sun04b]、CORBA

组件模型（CCM）[OMG04d]和COM+[Ewal01]。它们都实现了RLM。类似地，Java 2 Connector Architecture（JCA）[Sun04i]定义了把用户定义的RLM与基于EJB的RLM整合的框架。

远程中间件。中间件技术（比如CORBA [OMG04a]和.NET Remoting [Ramm02] 在多个层次实现了RLM。中间件确保对连接、线程、同步原语、远程服务之类的资源的正确生命周期管理。

目前的发展（比如Ice[Zero04a]）表明，不是只有CORBA和.NET才是实现了RLM功能的中间件框架。RLM是所有中间件框架中都存在的已经被证明的概念。

网格计算[BBL02] [Grid04]。网格计算的内容是关于共享和聚合分布式资源（比如处理时间、存储器和信息）。网格包含了多台连接到一起成为一个系统的计算机。参与的计算机为网格提供资源，它们必须通过一些手段管理自己的本地资源。通常会用资源生命周期管理器来使得网格中的资源可用，并满足本地和分布式的资源请求。

又见（See Also）

Object Lifetime Manager [LGS01]模式专门用来管理Singleton对象，比如不支持正确的静态析构的操作系统（例如实时操作系统）中的资源。

Garbage Collector模式[JoLi96]专门用于清除不用的对象以及它们的相关资源。因此，垃圾收集不是完整的RLM，因为它不处理资源的创建、分配或者获取。

Pooling模式专注于资源的循环利用，所以覆盖了资源的完整生命周期。

Caching模式专注于避免昂贵的资源获取以及相关联的初始化操作。

Manager模式[Somm98]专注于对象的管理，而不是一般的资源管理。

Abstract Manager模式[Lieb01]专注于企业系统中的业务对象的管理，而不是一般的资源管理。

Resource Exchanger模式[SaCa96]描述了资源如何在资源使用者之间共享。它使用产生器（比如网络驱动器）和接收器（比如服务器）。一开始产生器获取资源，然后同接收器交换。这给资源置了一个状态，稍后产生器会读取这个状态。这个模式通过使用Resource Exchanger管理generator生成的资源来保持资源的使用情况稳定。

致谢（Credits）

感谢我们的EuroPLoP 2003审稿人Ed Fernandez，以及参与笔者研讨会的Frank Buschmann、Kevlin Henney、Wolfgang Herzner、Klaus Marquart、Allan O’Callaghan和Markus Völter。

PDG

第4章 资源释放

“我看到天使被雕刻在大理石中，直到我把他释放。”

——Michelangelo Buonarroti

概览

及时释放不再用到的资源对于维护系统稳定性并避免资源枯竭的情况是至关重要的。优化资源的释放会带来好处，因为系统性能和可伸缩性直接依赖于它。如果可重用的资源不得不反复获取，那么就会造成额外的开销。如果当资源不再需要时没有被释放，那么可能就会导致资源“饥荒”。这又会导致系统不稳定并降低系统的可伸缩性。

显式地释放资源可能会比较麻烦，而且有时候容易导致错误。Leasing模式和Evictor模式都涉及了资源释放，它们可确保获取的可重用资源被及时释放。Leasing模式通过在获取资源时把资源同基于时间的租约相关联而简化了资源释放。当租约过期并且没有被更新时，那么资源就会被自动释放。Evictor模式则解决了决定何时释放资源以及释放哪些资源的问题。这两个模式都有助于优化在任何时间获取资源的数目。

4.1 Leasing模式

Leasing（租约）模式通过在获取资源时把资源同基于时间的租约相关联而简化了资源释放。当租约过期并且没有被更新时，那么资源就会被自动释放。

实例（Example）

考虑一个由多个分布式对象（用CORBA[OMG04a]实现）组成的系统。为了让客户可以访问这些分布式对象，容纳对象的服务器通过CORBA Naming Service[OMG04c]之类的查找服务发布对象引用。然后客户就可以通过查询查找服务来获得对象引用。例如，考虑一个注册到CORBA Naming Service的分布式报价服务对象。报价服务向任意连接它的客户提供股票报价信息。客户查询Naming Service以获得到报价服务的引用，然后直接同报价服务通信以获得股票价格（见图4-1）。

考虑一下，如果容纳报价服务的服务器崩溃了并且再也无法恢复，将会发生什么。报价服务将无法再访问，但是它的引用永远都不会从Naming Servie删除。这会带来两个问题：首先，客户还是可以从Naming Service获得报价服务的引用。但是，引用是无效的，因此客户发送的任何请求通常都会导致抛出异常。其次，缺少任何显式的方法来删除失效的对象引用，于是过一段时间之后，失效的对象引用这样的不使用的资源将在查找服务中堆积。

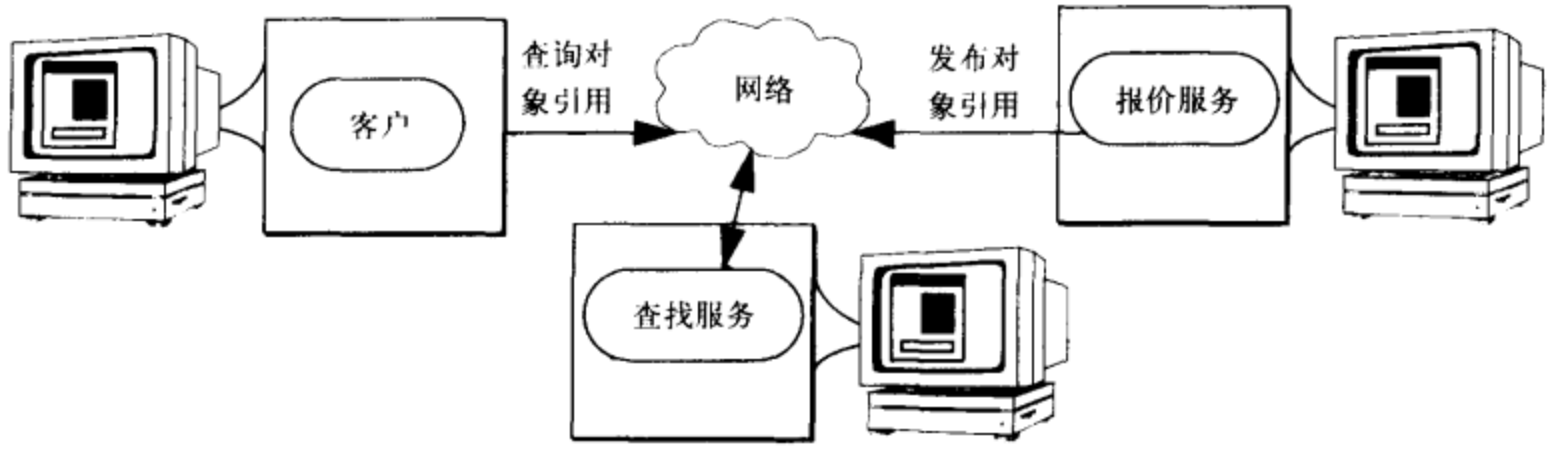


图 4-1

背景（Context）

需要控制资源使用，以允许及时释放不使用的资源的系统。

问题（Problem）

高度健壮和可伸缩的系统必须高效地管理资源。资源可以有很多种类型，包含本地和分布式服务、数据库会话以及安全令牌。在典型的用例中，资源使用者获取资源提供者的接口，然后向提供者要求一个或者多个资源。假定提供者给予了资源，那么资源使用者就会开始使用资源。但是，过了一段时间之后，资源使用者可能不再需要其中的一些资源。除非资源使用者显式地终止和提供者的关系并释放资源，否则不用的资源依然会被不必要地占用。这会进而导致资源使用者和提供者的性能下降，还会影响资源对其他资源使用者的可用性。

在资源使用者和资源提供者都是分布式的系统中，可能过了一段时间之后提供者所在的机器会崩溃，或者提供者不再提供某些资源。除非资源使用者显式地被提示资源不再可用，否则的话资源使用者可能会继续引用无效的资源。

所有这些导致的结果就是资源使用者这边可能有一些资源永远都不会被释放。这个问题的一个解决方案是使用某种监控工具，间歇性地检查用户的资源使用情况，以及资源使用者使用的资源的情况。这个工具可以建议资源使用者释放的资源。但是，这一方案既麻烦又容易出错。而且，监控工具也不利于性能。

为了以有效且高效的方式解决这一问题，需要解决下面的作用力：

- 简单性（Simplicity）。对资源使用者而言，资源管理应该简单，可以让资源使用者可选地释放不再需要的资源。
- 可用性（Availability）。资源使用者不再使用的资源，或者不再可用的资源，应当立刻被释放，以便让它们可以被新的资源使用者使用。例如，同网络连接关联的资源应该在连接断开之后就被释放。
- 优化（Optimality）。不使用的资源造成的系统负担应该最小。
- 真实性（Actuality）。当有新的资源可用时，资源使用者不应该使用资源的过期版本。

解决方案（Solution）

为每个被资源使用者使用的资源引入一个租约。授予者提供租约，持有者获得租约。租约

授予者通常是资源提供者；租约持有者通常是资源使用者。租约说明了资源使用者可以使用资源的时间长度。

如果资源被资源使用者所持有，那么一旦超过了时间期限，我们就说租约过期了，相应的资源就被资源使用者释放。另一方面，如果资源被资源提供者所持有，而资源使用者持有到资源的引用，那么租约过期的时候资源的引用就变得无效，并由资源使用者释放。此外，资源提供者也可以释放资源。

在租约有效的时候，租约持有者可以取消租约，在这种情况下相应的资源也被释放。在租约过期之前，租约持有者可以向租约授予者申请延长租约。如果租约被延长，那么相应的资源继续可用。

结构（Structure）

下列参与者形成了Leasing模式的结构:

- 资源提供某种类型的功能或者服务。
- 租约（Lease）提供一种时间的记法，可以同资源的可用性相关联。
- 授予者（Grantor）授予资源的租约。租约的授予者通常就是资源提供者。
- 持有者（Holder）获得资源的租约。租约的持有者通常就是资源使用者。

下面的CRC卡片描述了参与者的职责与协作（见图4-2）。

Class Resource	Collaborator
Responsibility • 代表可重用实体、比如连接或者服务	

Class Lease	Collaborator
Responsibility • 规定资源可用的时间期限	

Class Grantor	Collaborator
Responsibility • 把资源的租约授予持有者	• Holder • Lease • Resource

Class Holder	Collaborator
Responsibility • 获取并维护租约 • 使用资源 • （可选地）更新租约	• Resource • Lease • Grantor

图 4-2

下面的类图描述了Leasing模式的这些参与者之间的相互依赖关系（见图4-3）。

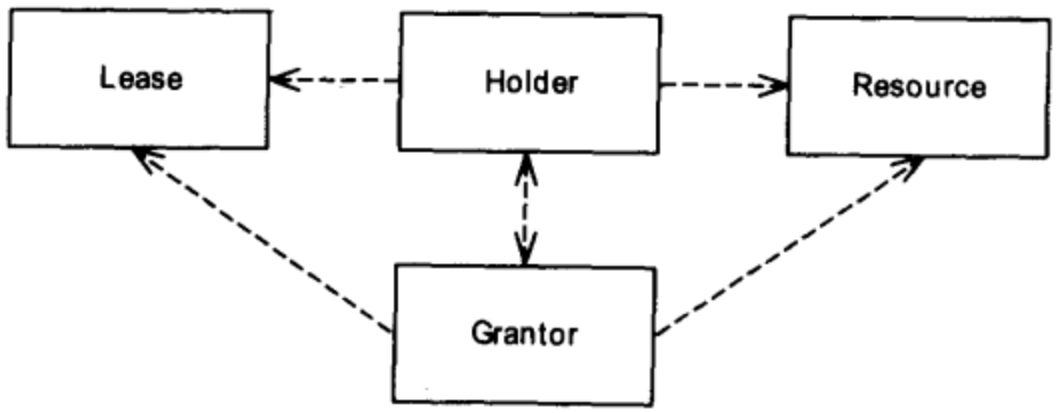


图 4-3

动态（Dynamics）

在下面的顺序图中，租约的授予者就是资源提供者，租约的持有者就是资源使用者。

场景1：第一步，资源使用者先从资源提供者处获得资源。在这一步中，租约被创建，租约请求者和租约授予者协商租约的长度。资源使用者获得资源及相应的租约。从那时开始，资源使用者可以访问资源（见图4-4）。

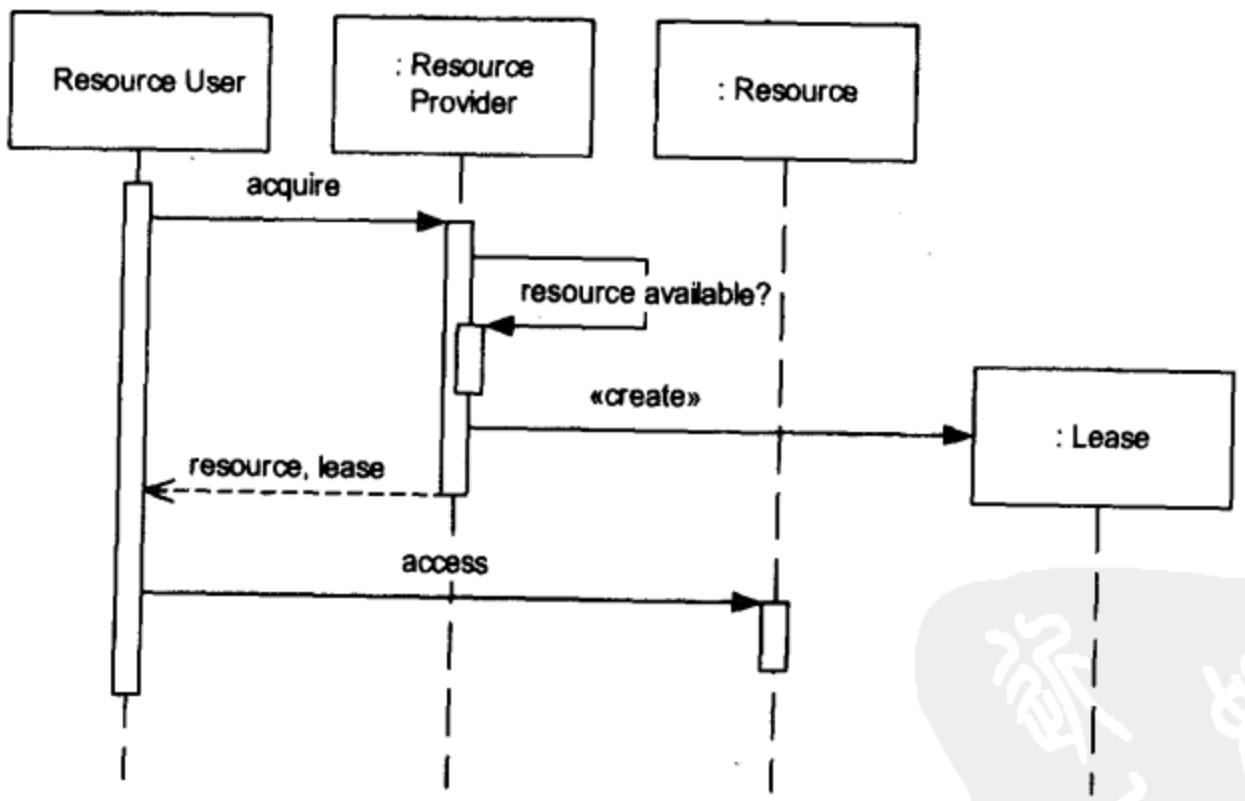


图 4-4

场景2：下面的顺序图展示了租约过期，资源提供者通知资源使用者的场景（见图4-5）。

响应时，资源使用者有多种选择。例如，资源使用者可以通过和资源提供者交互试图延长租约。顺序图的第一部分展示了这一场景。如果租约是一个主动实体，它也可以自行延长。参见“变体（Variants）”一节以获得更多细节。

如果租约不需要再延长，那么资源使用者可以接受其过期，之后资源使用者可以释放资源。顺序图的第二部分展示了这一场景。

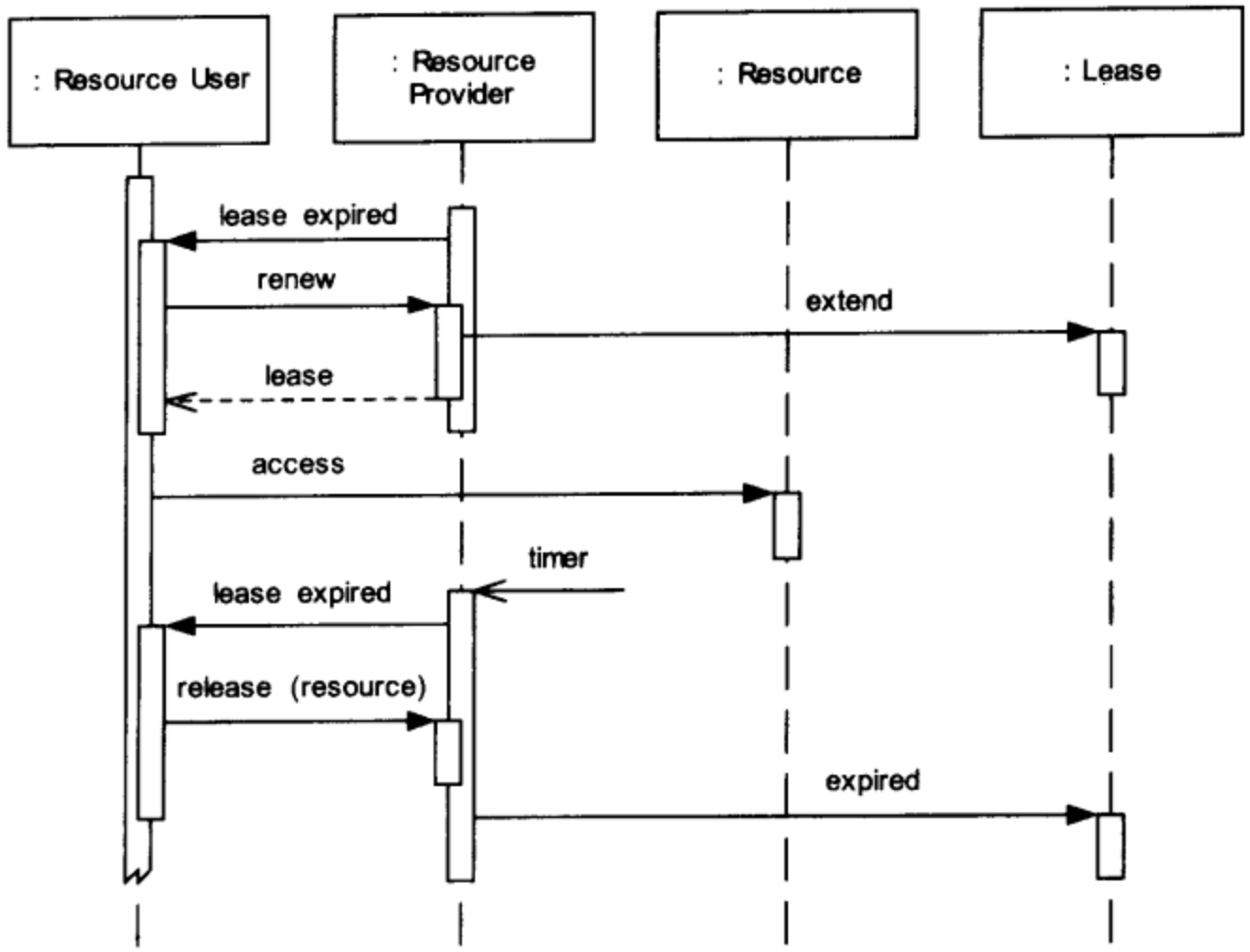


图 4-5

如果租约是一个主动实体，那么它也能使用Evictor模式发起资源释放过程。参见“变体(Variants)”一节以获得更多细节。

场景3：在资源被资源提供者持有，资源使用者只持有引用的情况下，租约过期时的交互略有不同。当租约过期时，资源使用者只需要释放资源引用，实际资源则由资源提供者来释放。下面的顺序图描述了这一情形（见图4-6）。

实现（Implementation）

实现Leasing模式涉及以下几步：

1) *决定要关联租约的资源*。所有可用性基于时间的资源都可以关联租约。这包含了生命短暂的、不连续使用的资源，以及会频繁被新版本更新的资源。

2) *决定租约创建策略*。租约授予者会为每个使用的资源创建租约。如果多个资源使用者共享一个资源，那么会为资源创建多个租约。租约授予者可以使用工厂来创建租约，如同Abstract Factory模式[GoF95]所描述的那样。租约创建时要求说明租期。租期可以取决于资源的类型、请求的租期以及租约授予者的策略。租约请求者和租约授予者可以就将授予的租约的租期进行协调。

资源使用者可能会想把资源以及相关的租约传递给另一个资源使用者。可以用租约创建策略来说明是否可以这样做。如果资源及其相关联的租约可以被传递给其他资源使用者，那么租约就需要提供操作以便让它的所有关系可以改变。

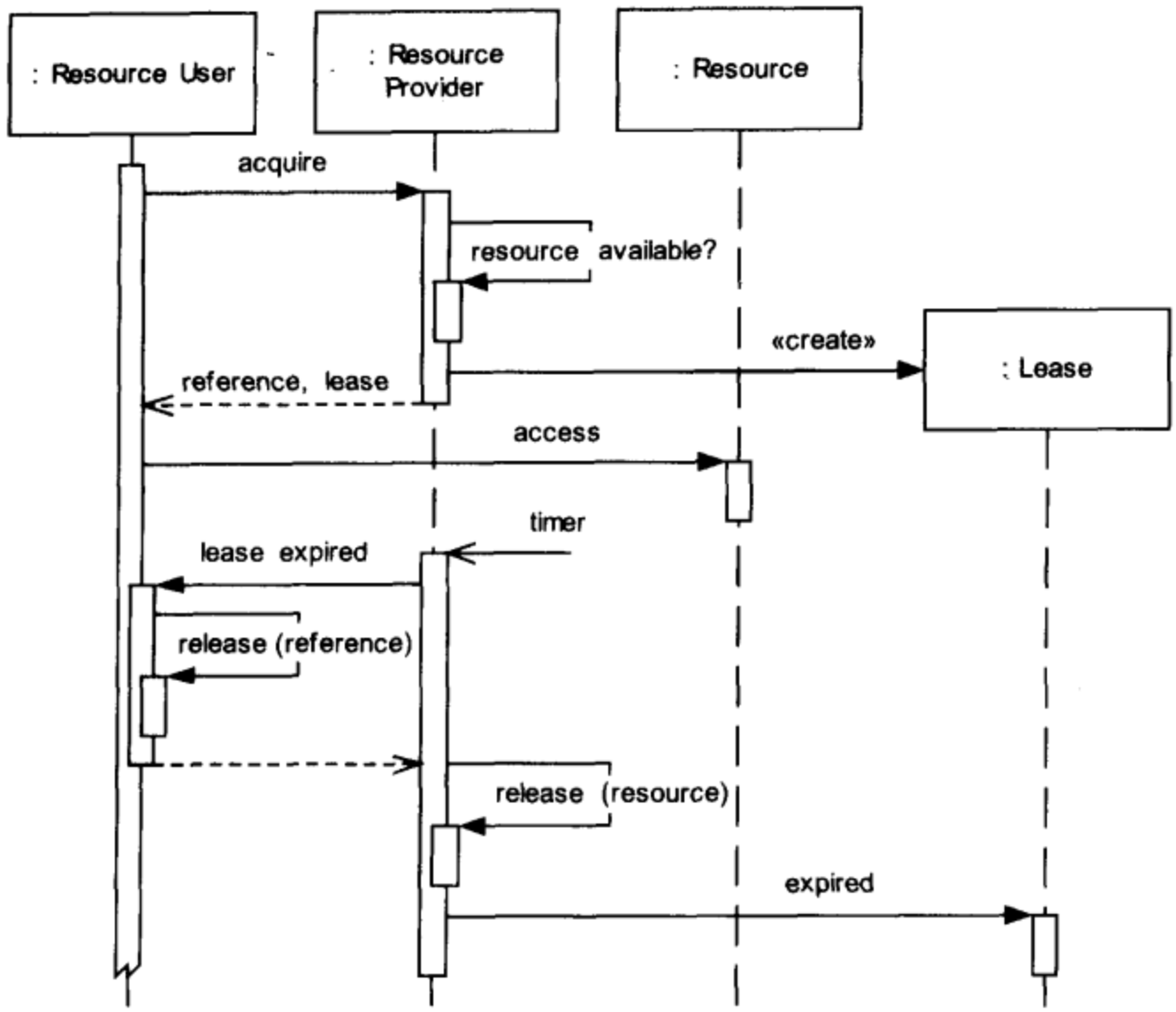


图 4-6

3) 提供租约持有者功能。每个租约持有者都必须实现一个通告接口，租约授予者可以通过这个接口来通知租约持有者租约过期。当资源使用者从租约授予者那里获取了一个资源及其对应的租约时，它将会把这个接口注册到租约授予者。

在C++中，这样的接口可能看起来像这样:

```
class LeaseHolder
{
public:
   void lease_expired (Lease &lease) = 0;
};
```

这个lease_expired()方法是租约授予者用来通知租约持有者租约过期的回调方法。在这个方法的实现中，资源使用者可以释放租约过期的资源。

4) 提供租约授予者功能。一旦租约被创建，授予者就需要维护租约和相应资源之间的映射。这使得授予者可以跟踪资源被使用的时间，并且决定对此资源是否依然可以授予新的租约。

此外，为了支持对资源使用者的通告，租约到相应资源使用者的映射也是必要的。Observer模式[GoF95]描述了如何实现这样的通告。

5) 决定租约职责。如果租约可以被更新，那么有必要决定由谁来负责更新。如果租约是一

个活动实体，那么它可以自动更新自己，否则的话更新过程可能就需要授予者与持有者重新协调。对租约的重新协调可能会导致租约的新策略，包括租约更新的时间。

6) 决定租约过期策略。一旦租约过期并且没有更新，那么同它关联的资源就需要被释放。这可以自动进行，比如使用Evictor模式，或者要求资源使用者的干预。类似地，租约授予者需要删除租约、资源和资源使用者之间的映射。通常租约会包含某种携带关于它的持有者信息的Asynchronous Completion Token [POSA2]，以便可以在租约过期时正确地完成清除工作。

实例解析（Example Resolved）

考虑这个例子，分布式报价服务对象需要对CORBA客户可用。包含报价服务对象的服务器把对象注册到一个CORBA Naming Service。因此，这个Naming Service是资源提供者，资源是注册到Naming Service的服务对象引用。包含报价服务的服务器是资源使用者。服务器和Naming Service会协商租约细节，包括租约对象引用需要注册的租期长度，以及关于租约更新的策略。一旦完成了这些协商，Naming Service就会注册报价服务对象引用，并且创建具有约定好的租期的租约。Naming Service是租约授予者，服务器是租约持有者。

在租约没有过期时，Naming Service会保持报价服务对象的引用，并让其可以被任何要求它的客户所获得。一旦租约过期，那么服务器就需要显式地更新租约，以表明希望让报价服务对象引用继续对客户可用。如果服务器不更新租约，那么Naming Service就会自动删除报价服务对象引用，并释放所有同它相关的额外资源。

下面的C++代码展示了服务器如何把报价服务对象引用注册到Naming Service。在这个例子中，LookupService为Naming Service提供了一个Wrapper Facade[POSA2]，所以CORBA Naming Service的标准接口不需要改变。LookupService扮演了租约授予者的角色。

为了发布报价服务，应用程序把其提供的报价对象绑定到查找服务。当租约时间被接受之后，会返回一个有效的租约。在租约失效前可以通过Lease接口更新它。当租约失效之后，会通知LeaseHolder类型的注册对象。

```
int main (int argc, char* argv[])
{
  // First initialize the ORB
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

  // Create a quoter service
  quoter _Impl *quoter _impl = new quoter _Impl;

  // Get the CORBA object reference of it
  quoter _var quoter = quoter _impl->_this ();

  // Create a holder callback object
  LeaseHolder_Impl *lease_holder_impl =
      new LeaseHolder_Impl;

  // Get the CORBA object reference of it
  LeaseHolder_var lease_holder = lease_holder_impl->_this ();

  // Get hold of the lookup service which is also the lease grantor
  CORBA::Object_var obj =
      orb->resolve_initial_references("LookupService");
```

```
// Cast from CORBA::Object
LookupService_var lookup_service = LookupService::_narrow (obj);

TimeBase::TimeT duration = 120; // seconds

CosNaming::Name name;
name.length (1);
name[0].id = "quoter ";

Lease_var lease;
try
{

    // Publish the quoter service by registering the object
    // reference of the quoter object with the lookup service
    lease = lookup_service->bind (name, quoter , duration,
                                  lease_holder);
}
catch (InvalidDuration &)
{
    // Try with a shorter lease duration or give up.
}

// Do other things
// ...

// Renew lease if still interested in publishing the
// quoter service
    duration = 180; // seconds
    lease->renew (duration);

    return 0;
}
```

下面的代码展示了LookupService_Impl类如何封装CORBA Naming Service。在检查可接受的期限并建立相应的租约之后，实际的绑定被委托给标准的Naming Service。请注意，使用了Reactor模式[POSA2]来实现计时器。因此，LookupService_Impl类不仅实现了LookupService接口，也实现了Event_Handler接口。这使得租约可以将自身注册到Reactor，以在租约到期时收到通知。

```
class LookupService_Impl :
   public POA_LookupService, Event_Handler {
public:
   LookupService_Impl () {};
   ~LookupService_Impl () {};

   Lease_ptr bind (const CosNaming::Name &name,
                   CORBA::Object_ptr object,
                   TimeBase::TimeT duration,
                   LeaseHolder_ptr lease_holder)
       throw (InvalidDuration)
   {
       if (this->duration_is_acceptable (duration))
       {
           // Create a new lease
           Lease_Impl *lease_impl =
```

```
            new Lease_Impl (current_time () + duration,
                            this->_this(), name, lease_holder);

        // Get the CORBA object reference
        Lease_var lease = lease_impl->_this ();

        reactor_->register_timer (duration, this, lease);

        // Delegate binding to actual Naming Service
        name_service_->bind (name, object);

        return lease;
    }
    else
    {
        // Reject the bind request
        throw InvalidDuration();
    }
}

void cancel_lease (Lease_ptr lease)
{
    CosNaming::Name_var name = lease->get_name ();

    // Delegate to the Naming Service
    name_service_->unbind (name);

    // Cancel timer with the reactor
    reactor_->unregister_timer (this, lease);
}

TimeBase::TimeT prolong_lease (Lease_ptr lease,

    TimeBase::TimeT duration)
    throw InvalidDuration()
{
    if (this->duration_is_acceptable (duration))
    {
        // Reschedule the timer with the reactor for the
        // new duration of the lease
        reactor_->unregister_timer (this, lease);
        reactor_->register_timer (duration, this, lease);
    }
    else // Reject the bind request
    {
        throw (InvalidDuration());
    }
}

// The lease is used as Asynchronous Completion Token [POSA2]
void on_timer_expire (Lease_ptr lease)
{
    LeaseHolder_var lease_holder =
        lease->get_lease_holder ();

    // Notify lease holder of lease expriation
    lease_holder->lease_expired (lease);
```

```
        if (lease->get_remaining_time () <= 0)
        {
            // If the lease did not get renewed by the lease holder
            lease->expired ();
            CosNaming::Name_var name = lease->get_name ();
            name_service_->unbind (name);
        }
    }

    //...
};
```

下面的代码展示了如何实现租约。查找服务会创建租约。它维护了用于注册的名字和租约持有者之间的联系。

```
class Lease_Impl : public POA_Lease
{
public:
   Lease_Impl (TimeBase::TimeT expiration_time,
               LookupService_ptr lookup_service,
               const CosNaming::Name &name,
               LeaseHolder_ptr lease_holder)
   : lookup_service_ ( LookupService::_duplicate (lookup_service)),
       lease_holder_ (LeaseHolder::_duplicate (lease_holder)),
       expiration_time_ (expiration_time),
       name_ (name),
       valid_lease_ (TRUE)
   {
   }

   // Renew the lease for the given time
   void renew (TimeBase::TimeT duration)
       throw (InvalidDuration)
   {
       TimeBase::TimeT accepted_duration =
           lookup_service_->prolong_lease (this->_this (), duration);

       if (accepted_duration != 0)
       {
           expiration_time_ = current_time () + accepted_duration;
           valid_lease_ = TRUE;
       }
   }

   TimeBase::TimeT get_remaining_time ()
   {
       if (valid_lease_)
           return expiration_time_-current_time ();
       else
           return 0;
   }

   void expired ()
   {
       valid_lease_ = FALSE;
   }
```

```
    LeaseHolder_ptr get_lease_holder ()
    {
        return LeaseHolder::_duplicate (lease_holder_);
    }

    CosNaming::Name *get_name ()
    {
        return &name_;
    }

    //...
};
```

在报价对象注册的时候，会传一个LeaseHolder_Impl类的对象到查找服务。当租约到期时，它会收到通告。它可以延长租约，也可以准备让报价对象从查找服务撤销注册。

```
class LeaseHolder_Impl : public POA_LeaseHolder
{
public:
   void lease_expired (Lease_ptr lease)
   {
       // prolong the lease
       TimeBase::TimeT new_duration = 240; // seconds
       lease->renew (new_duration);

       // or remove the binding of the quoter
     , // object from the lookup service
   }
   //...
};
```

变体（Variants）

特定的租约创建和过期策略可以带来Leasing模式的多种变体。

Active Lease（主动租约）模式。当租约是一个主动实体时，租约可以担负起通知租约持有者租约过期的职责。它可以直接释放资源，也可以使用Evictor模式来触发资源的释放。

Auto-renewable Lease（自动更新的租约）模式。可以用“当过期时自动更新自身”的策略来创建主动租约。在这种情形下，租约维护足够的关于持有者和授予者的信息，从而可以更新自身。具有较短租期的自动更新的租约要比单一的、比较长时间的租约好，因为每次租约更新都为租约持有者提供了更新其持有的资源的机会（如果资源发生变化的话）。这种模式进一步的变化是基于租约授予者和租约持有者之间的一些协调来限制自动更新的次数。

Leasing Delegate（租约代理）模式。租约的更新不需要由租约或者租约持有者自动完成，而是可以由单独的对象完成。这让租约持有者可以从当租约过期后更新租约的职责中解脱出来。

Non-expiring Lease（不会过期的租约）模式。可以建立不会过期的租约。在这种情况中，持有者必须在不再需要同租约关联的资源时显式地撤销租约。但是，这一变体失去了使用Leasing模式的很多好处。不过它可以用来集成不容易引入租约表示法的遗留系统。

Self-reminding Leases（自我提醒的租约）模式。可以用回调函数来提醒租约持有者租约过期了，这样它们就有机会更新租约。这对不想或者无法判断租约是否过期的租约持有者很有

帮助。

Invalidating Leases（失效租约）模式。Leasing模式使得失效的资源（比如对象引用）可以被及时释放。这个模式可以用Invalidation模式[YBS99]来扩展，从而使我们可以显式地释放失效的资源。如果资源失效了，那么资源提供者可以向租约授予者发出一个失效信号，租约授予者可以把信号传递给所有的租约持有者。租约持有者就可以撤销租约，使得资源被释放。请注意，Invalidation模式可能会带来额外的复杂性以及资源提供者、租约授予者和租约持有者之间的依赖性。因此，应该只在等待租约到期显得不够高效，有必要在资源失效后立即释放的情况下才使用它。

结果（Consequences）

使用Leasing模式有几个**优点**：

- 简单性（Simplicity）。Leasing模式为资源使用者简化了资源管理。一旦资源的租约过期，资源使用者没有更新它，那么就可以自动释放资源了。
- 使用效率（Efficiency of use）。资源提供者可以更有效率地控制资源的使用。通过把资源的使用与基于时间的租约相绑定，资源提供者可以通过尽快释放不使用的资源来避免浪费，这样就可以把资源交给新的资源使用者。这可以把不用的资源带来的系统开销最小化。
- 版本控制（Versioning）。Leasing模式使得资源的旧版本可以相对容易地被新版本替换。资源提供者可以在租约更新的时候向资源使用者提供新版本的资源。
- 增强系统稳定性（Enhanced system stability）。Leasing模式有助于提高系统的稳定性，这是通过确保资源使用者不会访问失效的资源来做到的。此外，Leasing模式还有助于减少不用的资源的浪费，从而避免资源短缺。

使用Leasing模式有一些**缺点**：

- 额外的开销（Additional overhead）。Leasing模式要求为资源提供者提供给资源使用者的每个资源都创建一个额外的对象（租约）。为租约对象建立一个池并对不同的资源分配重用它们可以帮助缓解这个问题。此外，当租约过期的时候，资源授予者需要发送通知给资源使用者，这也导致了额外的开销。
- 额外的应用程序逻辑（Additional application logic）。Leasing模式要求应用程序逻辑支持“作为资源使用者和资源提供者之间的胶水”的租约概念。应用程序构架师需要牢记资源不是无限的，它们不是永远可用的。把重复的租借代码放入框架可以减轻编码复杂性。
- 定时监视器（Timer Watchdog）。资源提供者和资源使用者都需要能够判断资源何时过期。这就要求某种定时器机制的支持，这在一些遗留系统中可能做不到。但是，如果遗留系统是基于事件的应用程序，那么只需要很少的开销就能让它们可以感知定时器。

已知应用（Known Uses）

Jini[Sun04c]。Sun的Jini技术广泛地使用了Leasing模式。它以两种方式来使用。首先，它把每个服务都同一个说明了服务有效期（客户可以使用服务的时间）的租约相关联。一旦租约过期，并且客户没有更新租约，那么对应于租约的服务对这个客户就变得不可用。其次，它还在每次服务向Jini查找服务注册的时候关联一个租约对象。如果租约过期了，并且相应的服务没有

更新租约，那么服务就从查找服务中删除。

.NET Remoting[Ramm02]。在.NET Remoting中，远程对象是由Leasing Distributed Garbage Collector管理的。可以用3种方式更新租约：第一，每一个新的请求都会隐式地更新租约。第二，客户端和服务端可以用ILease接口的Renew()方法来显式地更新租约。远程对象可以通过实现ILease接口来影响它们的基于租约的生命周期管理。第三，远程对象也可以把租约管理传递给一个租约提供者（一个“赞助商”，必须实现ISponsor接口）。当租约过期之后，会要求“赞助商”延长租约。通常，租约对于客户激活的对象最有用，因为Singleton通常会被配置成永久存活。

软件许可证[Macr04]。软件许可证可以看做是软件和用户之间的租约。用户可以获得使用某个特定软件的许可证。许可证本身可以从许可证服务器之类的地方获得，并且通常在一定时间内有效。一旦过了这段时间，那么使用者就必须更新许可证，否则就不能继续使用软件。

动态主机配置协议[DHCP04]。DHCP的目的是让IP网络上的各个计算机可以从DHCP服务器获得它们的配置设置。背后的动机是减少管理大型IP网络的负担。以这种方式分配的最重要的信息是IP地址。在这个背景中，DHCP租约是DHCP服务器许可DHCP客户使用特定IP地址的时间。租约时间通常是由系统管理员设置的。请注意，DHCP客户本身会监控租约何时过期，而不是由DHCP服务器来触发。

文件缓存。有一些网络文件系统（比如SODA [KoMa95]）把租约用做NFS之类的传统协议的扩展，以确保分布式文件系统中缓存文件块的一致性，从而获得性能和可伸缩性方面的改善。

杂志/报纸的订阅。Leasing模式在现实世界中一个已知的使用例子是杂志和报纸的订阅。在这个例子中，订阅表示了租约，通常会在一段确定好的时间之后过期。订阅者必须更新订阅，否则订阅就结束。在一些情况下，订阅者会设定自动更新，例如通过提供银行账户信息或者信用卡信息。

基于Web的电子邮件账户。很多基于Web的电子邮件账户，比如MSN Hotmail [MSN04]或者GMX [GMX04]在一段时间未用之后都会自动变成未激活状态。在这种情况下，这段时间可以被看做租约，租约的更新要求使用电子邮件账户。

又见（See Also）

Leasing模式最常见的实现依赖基于时间的回调函数来通知租约过期。基于时间的应用程序通常会使用一个或者多个事件分发器，或者使用Reactor [POSA2]，比如Windows消息队列（Message Queue）、InterViews的Dispatcher [LiCa87]或者ACE Reactor [Schm02] [Schm03a]。事件分发器通常会提供可以注册事件处理器（比如定时器到期后会调用的定时器处理器）的接口。

在不包含事件循环的应用程序中，可以用Active Object模式[POSA2]来替换定时器处理。Active Object有它自己的线程控制，可以利用操作系统或者运行它自己的事件循环来通过回调函数在定时器到期时通知租约。

为了让租约对于资源使用者透明，可以利用Proxy模式[GoF95][POSA2]。资源代理可以处理租约更新、策略协商以及租约的撤销，而这些本来通常都要用户去做。CORBA智能代理（Smart Proxies）[Schm04]在CORBA中提供了合适的抽象，而C++中的对应物则是智能指针（Smart Pointer）。

对于并发的资源，资源提供者必须管理多个持有到同一资源的租约的资源使用者。引用计数[Henn01]是跟踪“没有任何资源使用者持有租约，可以清除资源”事件的典型方案。

致谢（Credits）

感谢我们的PLoP 2000审稿人Irfan Pyarali，以及参与笔者研讨会的Ralph Johnson、Rani Pinchuk、Dirk Riehle、Yonat Sharon、Michel Tilman、Hallvard Traetteberg和Martijn van Welie。

4.2 Evictor模式

Evictor（清除者）模式描述了何时以及如何释放资源以优化资源管理。这个模式让我们可以配置不同的策略来自动决定哪些资源应该释放，以及应该在什么时候释放这些资源。

实例（Example）

考虑一个网络管理系统，负责管理多个网络元素（见图4-7）。这些网络元素通常用拓扑树表示。拓扑树提供了网络基础设施中的关键元素的虚拟化层次表现。网络管理系统允许用户查看这样的树，并且可以获得关于一个或多个网络元素的细节。取决于网络元素的类型，它的细节可能对应于大量的数据。例如，复杂网络元素的细节可能包含了它的状态以及它的组件的状态的信息。

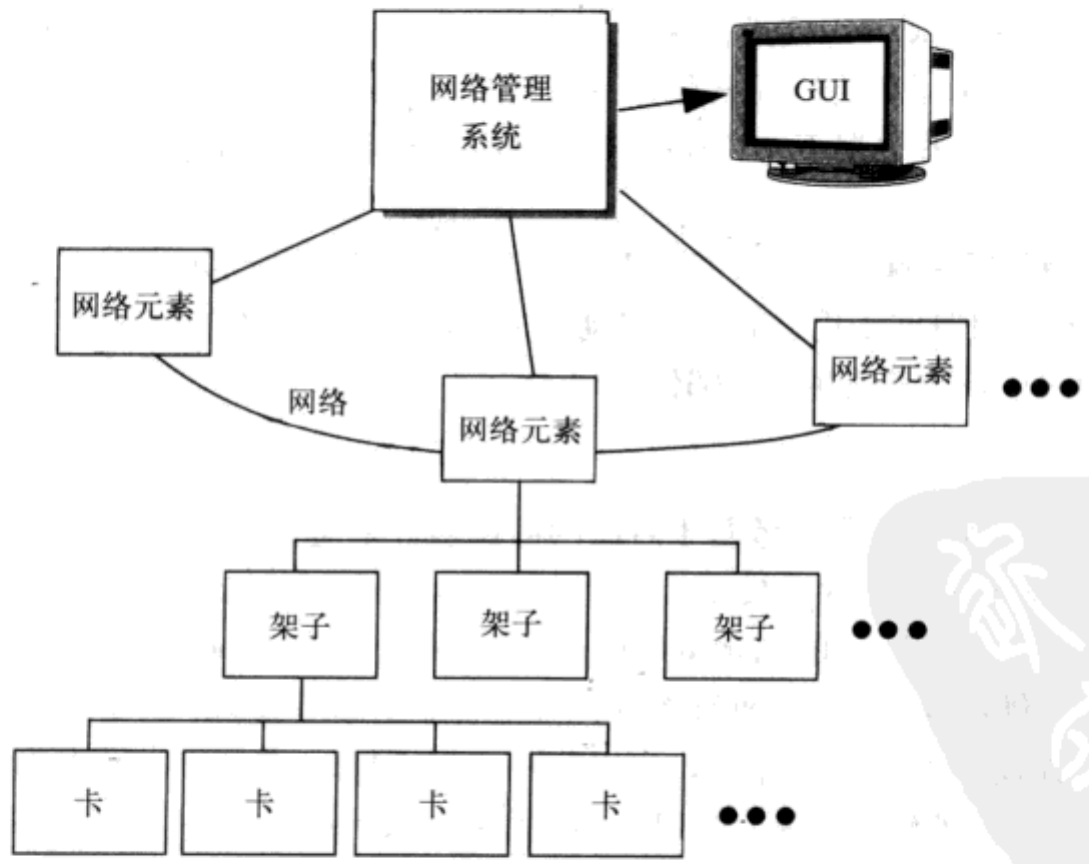


图 4-7

拓扑树通常会在应用程序启动时创建，或者当用户要求查看网络元素树的时候创建，或之间的某个时刻创建。类似地，获取所有网络元素的细节可以是在树创建的时候，也可以推迟到用户请求的时候。不管何时把网络元素的细节放入内存，在内存中过长地保存这些细节可能都

会代价高昂。如果一个网络元素的细节再也不会被用户访问，那么它们将无意义地消耗宝贵的内存资源。但另一方面，用户可能会请求同一个网络元素的细节，因此在内存中保存细节可以增进性能。如果一个网络元素的细节被用户频繁访问，而它又没有被缓存，那么就会导致昂贵的获取实际网络元素细节的调用。这会降低系统的性能。

一个重要的问题是：何时释放资源才合适，应该释放哪些资源？

背景（Context）

需要控制对资源的释放以确保对资源的高效管理的系统。

问题（Problem）

高度健壮及可伸缩的系统必须高效地管理资源。随着时间的推移，应用程序会获得很多资源，其中有一些只用过一次。如果应用程序持续地获取资源而不释放它们，那么就会导致性能下降和系统的不稳定。为了避免出现这样的情况，应用程序可以在使用资源之后立刻释放资源。但是，应用程序可能需要重新使用相同的资源，这就要求重新获得那些资源。重新获得资源这个操作本身可能代价高昂，所以应该避免，这可以通过把频繁使用的资源保留在内存中来做到。

为了解决这些相互冲突的资源管理需求，我们需要解决下面的作用力：

- 最优性（Optimality）。使用资源的频率应当影响资源的生命周期。
- 可配置性（Configurability）。应当通过资源类型、可用内存和CPU负载之类的参数来决定资源的释放。
- 透明性（Transparency）。解决方案应该对资源使用者是透明的。

解决方案（Solution）

监控资源的使用，并用某种策略，比如Least Recently Used（LRU）或者Least Frequently Used（LFU）来控制其生命周期。每当资源被使用时，就会被应用程序所标记。当资源最近没有被使用，或者不是频繁被使用时，那么就不会被标记。周期性地，或者根据需要，应用程序会选择那些没有被标记的资源并且释放或者清除它们。被标记的资源会继续留在内存中，因为它们会被频繁使用。

另外，也可以用其他策略来判断应该清除哪些资源。例如，对于内存受限的应用程序，可以用资源的尺寸来决定应该清除哪些资源。在这样的情形下，消费大量内存的资源可能会被清除，哪怕它曾被频繁使用。

结构（Structure）

下面的参与者形成了Evictor模式的结构：

- 资源使用者使用资源，它可以包括应用程序或者操作系统。
- 资源是一个实体，比如提供某种类型的功能的服务。
- 清除者（Evictor）基于一个或多个清除策略清除资源。
- 清除策略（Eviction Strategy）描述用于判断是否应当清除资源的标准。

下面的CRC卡片描述了参与者的职责与协作（见图4-8）。

Class Resource User	Collaborator • Resource
Responsibility • 获取并使用资源	

Class Resource	Collaborator
Responsibility • 代表可重用实体，比如内存或者线程	

Class Evictor	Collaborator • Resource • Eviction Strategy
Responsibility • 基于一个或者多个清除策略清除资源	

Class Eviction Strategy	Collaborator
Responsibility • 描述用于决定清除哪个资源的标准	

图 4-8

Evictor模式的参与者形成了下面的类图（见图4-9）。

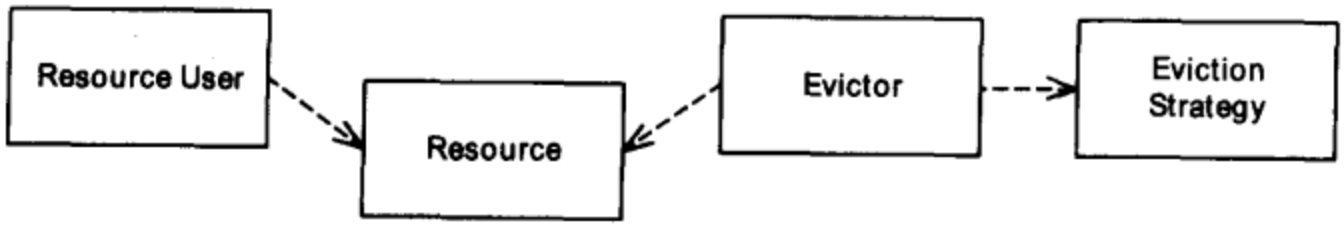


图 4-9

动态（Dynamics）

资源使用者首先获取资源。依赖于资源，资源使用者将其注册到Evictor。从此时开始，Evictor监视资源。当Evictor检测到资源根据清除策略应该被清除掉时，它就询问资源，判断是否应该清除。如果资源可以被清除，那么就会得到“处理后事”的机会，之后就被删除（见图4-10）。

在大多数实现中，步骤的顺序是一样的。但也会有一些变体，比如资源可能无法被标记，结果就是Evictor不得不截获到资源的请求以获得其清除策略所依赖的统计信息。

实现（Implementation）

实现Evictor模式涉及4步：

1）定义清除接口。应该定义一个清除接口，所有能被清除的资源都要实现这个接口。

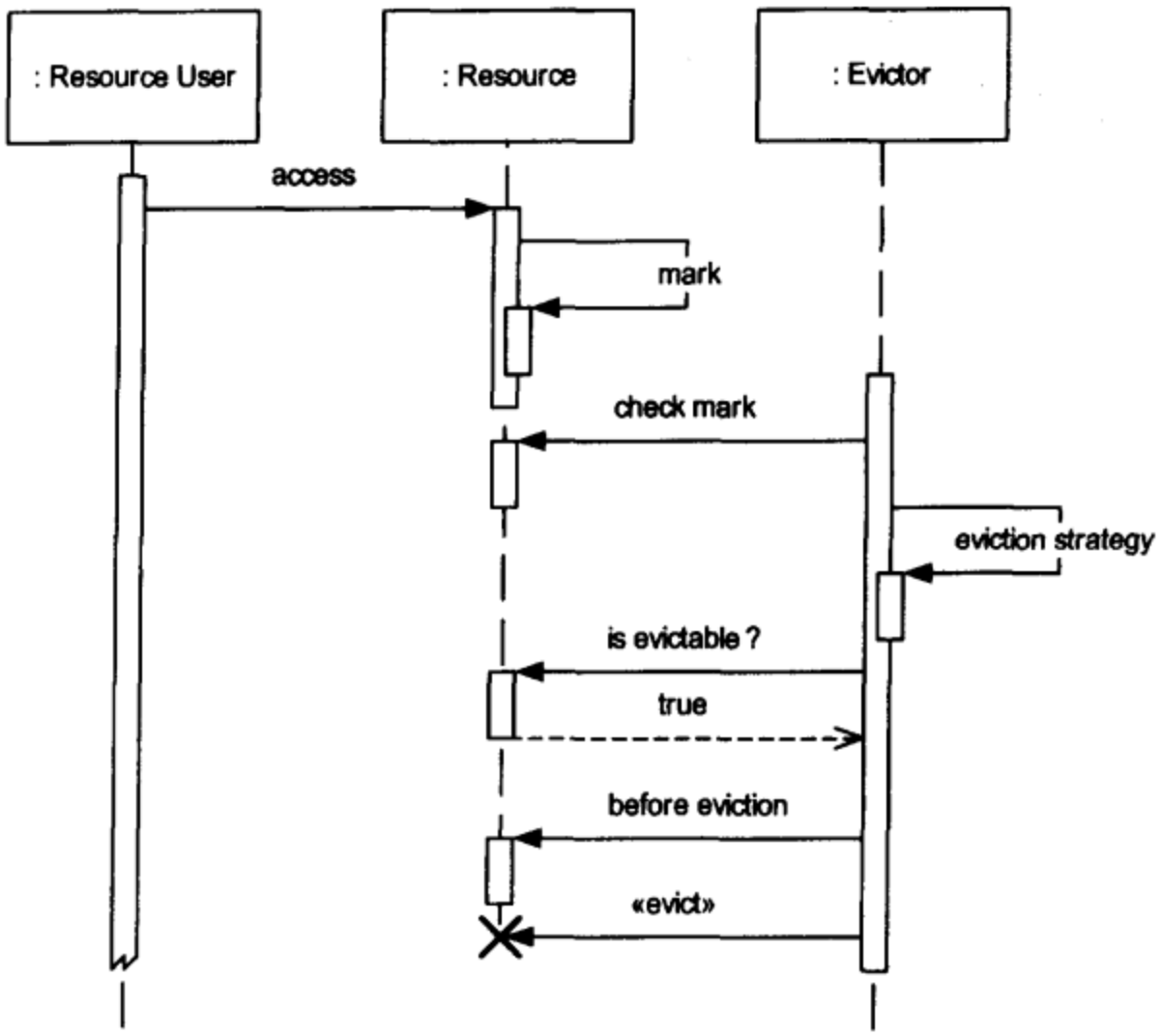


图 4-10

例如，在Java中清除接口可能看起来像这样:

```
public interface EvictionInterface {
   public boolean isEvictable ();
   public Object info ();
   public void beforeEviction ();
}
```

可以用isEvictable() 方法来判断对象是否可以被清除（详见第2步）。Evictor使用了info() 方法来从对象处获得同策略相关的信息，以便决定是否要清除该对象（详见第4步）。而beforeEviction() 则是一个钩子方法[Pree94]，会在Evictor清除对象之前调用。这就给了对象清除它可能获取到的资源的机会。

例如，EJB *Session Bean* [Sun04b]和Entity Bean接口包含了一个叫做ejbPassivate()的方法，会在Entity Bean或者Session Bean清除之前调用。这就给了Bean释放所有已经获得的资源的机会。

2) 决定可清除的资源。开发者必须决定哪些资源可以并且应该被清除。例如，应用程序总是需要的资源，或者不能重新获取的资源就不应该被清除。任何可以清除的资源必须实现清除接口。在清除资源之前，应用程序应该调用这个接口，给资源一个机会来做“善后”工作，这包括把任何必要的状态持久化。

在上面描述的Java接口中，应用程序可以用isEvictable()方法来表明资源是否可以被清除。

如果方法返回true，那么Evictor会认为可以考虑把资源清除。如果方法返回false，那么Evictor会忽略此资源。

3) *决定清除策略*。基于应用程序的需求，可以用不同的清除策略来判断是否清除资源，以及清除哪些可以清除的资源。一些用来判断清除哪些资源的策略包括Least Recently Used（LRU）和Least Frequently Used（LFU）。

此外，可以使用能接受不同参数的用户定义的策略。例如，清除策略可以考虑重新获取被清除的资源的代价有多昂贵。使用这样的策略，重新获取的代价比较低的资源可能会被清除，哪怕它们会比获取起来更昂贵的资源用得更频繁。

4) *定义系统中对清除的使用*。需要在Evictor中增加清除资源的业务逻辑。这包含判断应该何时以及如何清除资源，以及实际标记要被清除的资源。通常情况下，Evictor在应用程序中作为一个单独对象或者组件而存在，并会被应用程序以必要的清除策略来配置。例如，应用程序可能会选择在可用内存低于某个特定值的时候清除资源。另一个不同的应用程序可能会实现更主动的策略并会周期性地清除资源，即便内存没有低于某个值。

Evictor可以使用Interceptor模式[POSA2]来截获用户对对象的访问。Interceptor可以以对资源使用者完全透明的方式来把对象标记成最近正在被使用。Evictor会周期性地或者根据需要查询所有可清除的对象，以便决定如果需要清除对象的话应该清除哪一些。在前面描述的Java接口中，Evictor会对每个对象调用info()方法并使用获得的信息在清除策略的背景下判断是否应该清除这个对象。对于实际的清除，可以使用Disposal Method模式[Henn03]，有时它也叫做Explicit Termination Method模式[Bloc01]。Disposal Method模式描述了类应该如何提供显式的方法来允许它们在被清除之前执行“善后”工作。

实例解析（Example Resolved）

考虑网络管理系统的例子，这个系统负责管理多个网络元素组成的网络。网络元素的细节可以在系统启动时获取，也可以在资源使用者请求时再获取。事先不知道用户意图，所以需要优化资源使用，以便只让那些被资源使用者频繁访问的网络元素保留在内存中。所以，这个例子中使用的清除策略是，清除过了某段设定的时间依然没有被资源使用者访问的网络元素。

每个网络元素都需要实现Eviction接口：

```
public class NetworkElement implements EvictionInterface {
   private NetworkElementComponent [] components;
   private Date lastAccess;

   public boolean isEvictable () {
       // Assume all network elements can be evicted
       return true;
   }

   public Object info () {
       // Return the date/time of last access, which
       // will then be used by the Evictor to determine
       // whether or not to evict us
       return lastAccess;
   }
```

```
    public void beforeEviction () {
        // First, release all resources currently held

        // Now, call beforeEviction() on all network element
        // components to give them a chance to release
        // necessary resources
        for (int i = 0; i < components.length; i++) {
            components[i].beforeEviction ();
        }
    }
    // ... other network element operations ...
}
```

类似地，所有的网络元素组件和子组件都需要实现Eviction接口，这样它们被清除时就可以递归地释放所有资源。

Evictor可以实现为一个在自己的线程控制中运行的对象，这样它就能周期性地检查是否有网络元素过了一段特定时间依然没被访问。

```
public class Evictor implements Runnable {
   private NetworkElement [] nes;
   public Evictor () {
       new Thread(this).start();
   }

   public void run() {
       // For simplicity, we run forever
       while(true) {
           // Sleep for configured amount of time
           try {
               Thread.sleep(pollTime);
           }
           catch(InterruptedException e) { break; }

           // Assume "threshold" contains the date/time such
           // that any network element accessed before it will
           // be evicted

           // Go through all the network elements and see
           // which ones to evict
           for(int i = 0; i < nes.length; i++) {

               NetworkElement ne = (NetworkElement)nes[i];
               if (ne.isEvictable()) {
                   Date d = (Date) ne.info();
                   if (d.before(threshold)) {
                       ne.beforeEviction ();
                       // Now remove the network element
                       // application-specifically
                   }
               }
           }
       }
   }
}
```

请注意，info() 方法返回的信息以及Evictor解释信息的方法是同应用程序相关的，可以按照需要部署的清除策略来裁剪。

变体（Variants）

· Deferred Eviction模式。清除一个或多个对象的过程可以细化成两步的过程。不是立刻删除对象，而是可以把它们放进一个FIFO队列。当队列满了的时候，队列前面的对象就被清除了。因此，队列扮演了待清除对象的中间持有者的角色。这样就给了对象“第二次机会”，这和Caching模式有些类似。如果这些对象中的任一个在从队列删除前又被访问了，那么它们就不需要有创建和初始化的开销。当然，这一变体会带来维护队列的开销，而且把清除对象保持在队列中也会占用资源。

Evictor with Object Pool模式。可以用一个对象池来保存被清除的对象。在这个变体中，当对象被清除，它不是从内存中被完整地删除，而只是失去了标识，成了匿名对象。之后，这个匿名对象会被添加到对象池中（只要对象池还没有满的话）如果对象池满了，那么对象就会被从内存中删除。当需要创建新对象时，队列中的匿名对象就可以出队并且获得新的标识，这就降低了对象创建的开销。对象池的尺寸可以根据可用内存来设置。更多的细节请参见Pooling模式。

Eviction Wrapper模式。可以被清除的对象不需要直接实现Eviction接口。可以用实现了Eviction接口的Wrapper Facade[POSA2]来包含对象。Evictor会调用包装对象的beforeEviction()方法，这个方法负责清除实际的对象。这一变体使得整合遗留代码变得简单，不必要求现有的类实现Eviction接口。这个变体的一个例子是在Java中使用对象引用作为包装对象。关于这个例子，更多细节请参见“已知应用（Known Uses）”一节。

结果（Consequences）

使用Evictor模式有几个**优点**:

- 可伸缩性（Scalability）。Evictor模式允许应用程序对使用的资源的数目保留一个上限，从而在任何给定的时间内都在内存中。这使得应用程序可以伸缩而不影响总体内存消耗。
- 低的内存占用（Low memory footprint）。Evictor模式允许应用程序通过可配置策略来控制哪些资源应该保留在内存中，哪些资源应该释放。通过只在内存中保留最关键的资源，应用程序可以保持“苗条”，并保持高效。
- 透明性（Transparency）。使用Evictor模式使得资源释放对于资源使用者完全透明。
- 稳定性（Stability）。Evictor模式降低了资源枯竭的可能性，从而增加了应用程序的稳定性。

使用Evictor模式有几个**缺点**:

- 额外开销（Overhead）。Evictor模式需要额外的业务逻辑来判断要清除哪些资源，以及实现清除策略。此外，资源的实际清除也可能会带来明显的执行开销。
- 重新获取的损失（Re-acquisition penalty）。如果再次需要用到被清除的资源，那么就需要重新获取该资源。这可能代价高昂，会影响应用程序的性能。可以通过调整Evictor用来判断清除哪些资源的策略来避免发生这种情况。

已知应用（Known Uses）

Enterprise JavaBeans(EJB) [Sun04b]。EJB规约定义了一个activation和deactivation机制，可以被容器用来把bean从内存中换出到次级存储器，从而为其他需要激活的bean释放了内存。bean实例必须实现ejbPassivate()方法，并且释放所有获得的资源。这个方法会被容器在换出bean之前调用到。

.NET [Rich02]。.NET的Common Language Runtime（CLR）内部使用了垃圾收集器来释放不用的对象。垃圾收集器根据对象的寿命，分3步给对象归类。如果对象想要在完全清除之前被通知，那么它必须实现Finalize()方法。.NET建议使用Disposal Method模式[Henn03]的Dispose()方法，客户应该用这个方法来显式地释放对象。

Java [Sun04a]。JDK 1.2的发布引入了引用对象的概念，可以用来在堆内存不够或者对象不用的时候清除对象。程序可以用引用对象来维护到另一个对象的引用，而后者依然可以在内存不够的时候被垃圾收集器所收集。此外， Reference Objects API定义了称做reference queue（引用队列）的数据结构，在清除之前，Garbage Collector [NoWe00]在这个地方放置引用对象。当引用的特定对象变软、变弱，或者难以触到的时候（所以准备好了被清除），应用程序可以引用队列来让特定的对象执行必要的操作。

CORBA [OMG04a]。为了在应用程序中管理内存消耗，通常会有一个Servant Manager来保留实例化的服务者的数目上限。如果服务者的数目达到了规定的限制，那么Servant Manager可以清除一个实例化的服务者，之后再为当前需求实例化一个新的服务者[HeVi99]。

换页 [Tane01]。大多数支持虚拟内存的操作系统都使用了换页的概念。进程所用的内存会被划分成页面。当操作系统需要的内存不够时，不使用的页面就会从内存中清除，并写到磁盘文件上去。当一个页面被再次访问，那么OS（操作系统）就会把需要的信息从磁盘复制到主存。这使得OS可以限制主存中页面数目的上限。换页跟通常的交换不同：交换一次清除一个进程的所有页面，而换页则只是清除单独的页面。

又见（See Also）

Leasing模式描述了资源的使用如何同时间相挂钩，使得不使用的资源可以自动释放。Lazy Acuisition模式描述了如何在系统运行时的“最后一刻”（重新）获取资源，以便优化资源使用。

Resource Exchanger模式[SaCa96]描述了如何降低服务器分配和管理资源的负载。Evictor模式是针对资源的释放来优化内存管理，而Resource Exchanger 则是通过交换资源来减少资源分配的开销。

Credits

感谢西门子公司技术部的模式小组，感谢我们的PLoP 2001审稿人John Vlissides的宝贵反馈和意见。

PDG

第 5 章
资源管理准则

“首先检查设计是否聪明且恰当；确定之后就坚定地‘追随’你的设计，不要越过你界定的目标。”

——William Shakespeare

本书中展示的资源管理模式语言解决的是软件系统的领域独立的非功能需求。该模式语言可以用于几乎任意领域，并且对解决该领域中的资源管理问题很有效。但是，每个领域都有它自己的特定并且反复出现的，几乎该领域中的所有应用程序都需要解决的问题集和作用力。辨识出应用程序需要解决的问题集以及作用力通常是在应用程序的设计阶段完成的。对一个特定的问题可能有几种解决方案，但找出一个同时还有利于高效地管理资源的解决方案则需要付出额外的努力。这是本书中描述的资源管理模式语言最能提供帮助的地方。

为了更好地把资源管理模式语言应用于特定领域以及特定应用程序，可以遵守下面的步骤：

1) 识别关键组件。进行有效资源管理的最重要步骤之一是首先识别出系统的关键组件。系统的关键组件是那些消费最昂贵资源或者消费最多资源的组件。如果一种资源很少，或者获取/释放资源的代价很大，或者会导致多个资源使用者的争夺，那么这种资源可能很昂贵。识别关键组件通常需要领域专门知识，以及（或者）对系统构架和设计的洞察。常见的策略是首先分析系统可能和资源有关的的功能和非功能需求。用这些需求作为基础，解决这些需求的组件应当被进一步分析。系统的关键组件对系统功能可能至关重要，也可能并不重要。

仅仅对系统进行分析通常并不能揭示出所有的关键组件，还需要分析系统的运行时行为。因此，要对系统进行测量（profiling）以找出资源的集中使用，进行性能测量和稳定性分析。同时也可以识别出昂贵的资源和使用这些资源的组件。

2) 识别环境。应用程序的问题和作用力通常描述了模式可以应用的初始环境。模式的应用把初始环境转变成了一个新的环境，称做结果环境。结果环境通常会成为一组新的作用力和需求的初始环境，从而导致继续应用其他的模式。当把资源管理模式语言用于每个关键组件的时候，应当找出并使用应用系统的初始环境。

思考一下Lazy Acquisition模式。这个模式可用于资源获取和资源使用之间有很大的时间间隙的初始环境。应用这个模式之后的结果环境是，资源被尽可能晚地获取，晚到在使用前的最后一刻。这就把获取的资源的数目最小化了。结果环境还具有这样的缺点：延迟的资源获取可能会导致运行时的性能降低。

3) 找出“热点”。应当为每个关键组件都找出所有热点。热点是组件的一个部分，它会对组件的下列方面之一有影响：资源消耗、性能、可伸缩性以及稳定性。在资源消耗增加的情况下，作为第一步，所有动态消耗资源的组件都应当被标示为可能的热点，直到找到资源消耗增加的根本原因为止。

通常情况下，需要解决的作用力和标示出的热点类别会有紧密联系。识别热点可能很难，因为不能把系统的所有部分都标示为“热点”，总会有一些部分比其他部分更关键。对组件的不同视角常常会揭示关于可能的热点的新信息。可以使用多种评估和测量方法来更好、更完整地找出热点。

4) 应用模式。对找出的每一个热点，都可以基于热点所属的类别和环境，应用资源管理模式语言中的一个或者多个模式。应用模式应当会导致一个新的环境，这个环境可能可以解决一开始的作用力集合。但是，可能会剩下一些作用力尚未解决，或者会增加新的作用力。在这样的情况下，应该继续应用模式语言中的其他模式。

下面两章描述的案例应用了这些准则。我们识别出了系统的关键组件以及系统中最关键的作用力，然后在每个案例的解决方案部分描述了如何解决热点。

第 6 章
案例分析——自组网络计算

“凌波微步、根据规约开发软件都不难，如果水面和规约都是冻结的话。”

——Edward V Berard

重点

几年前，Sun公司引入了一种新技术，称做Jini [Sun04c]。Jini提供了平台独立的“即插即用”技术，并且通过允许服务不经预先规划、安装或者人的干预就可加入网络而支持自组网络计算(ad hoc networking)。自组网络计算基于自发地添加、发现和使用网络中的服务的原则。服务可以有很多种，比如简单的时间服务、PowerPoint演示服务、MP3播放器服务，等等。一旦服务注册到网络之后，Jini允许其他设备和使用者发现新的服务。以透明的方式做到这些而不需要人工干预的能力体现了“即插即用”技术的基本原则。

自组网络计算领域依然吸引了大量的研究和开发人员。近来，有一些关于如何用C++实现Jini概念的研究已经完成了。JinACE [KiJa04]产生的不是Java字节码，而是同平台相关的跨越网络的共享库。类似地，除了Jini之外，还有很多技术，比如UPnP [UPnP04]以及微软的.NET [Rich02]都是围绕自组网络计算的概念建立的。

虽然所有的自组网络计算或者依赖于平台，或者依赖于语言，但是它们具有共同的底层构架。在这一共同的构架下，各个系统的资源管理需求是类似的。因为在自组网络中资源不断地增加和删除，所以用有效的方式来管理这些资源是很重要的。

6.1 概览

假定你想为你的公司建立一个自组网络计算解决方案。这个方案应该允许你在网络上以分布式服务的形式透明地分布和获取应用程序。大多数自组网络通常都会包含运行某种访问分布式服务的软件的移动设备。事实上，下一代移动电话愈来愈多地会提供允许动态加载和执行服务的运行时环境[Sun04j] [Symb03]（见图6-1）。

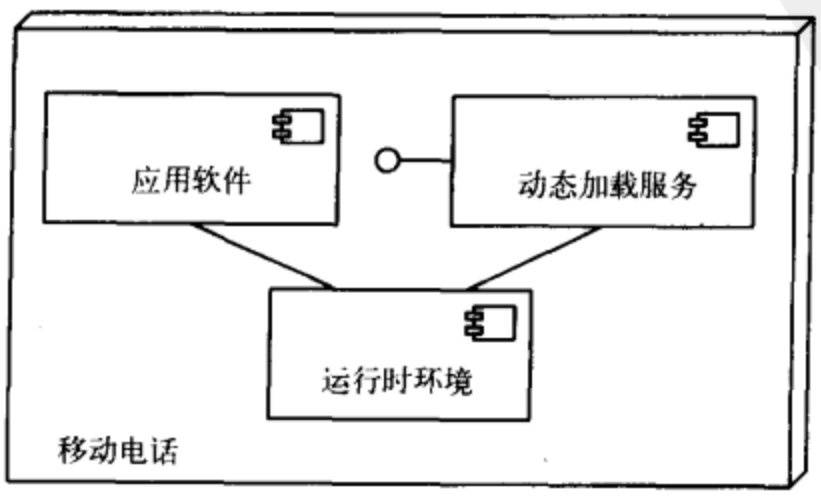


图 6-1

因为移动电话的资源（比如内存和处理器能力）有限，所以移动客户端的软件需要占用比较小的内存[NoWe00]。

作为一个例子，假定你的网络是一个移动网络，你的客户是移动电话。可以有很多种不同的服务，比如:

- 提供实时股票信息的服务。
- 提供最新新闻的服务，根据用户喜好而定制。
- 根据用户偏好和位置而发布广告的服务。

在这些情况中，服务本身位于移动电话中，并负责同后端提供者联系以获取必要的信息。但是，因为移动电话的处理能力和内存有效，所以不可能同时提供多个这样的服务。因此，为了节省移动电话的资源，服务需要根据需要来装载，当不再需要时就卸载。

如果需要根据需要来装载服务，那么必须有一个机制来动态地发现服务。服务应当在网络中可用，以便让移动电话能够轻而易举地发现它。一旦移动电话发现了服务，它应该可以下载和安装服务，而不需要用户交互。

随着时间的推移，如果用户不再需要或者不再使用一个或多个服务，那么应该有一种机制来卸载这些服务，以便为移动电话保留稀缺的资源。这也应该不需要任何用户干预。

6.2 动机

客户的有限资源和高度动态的行为要求灵活的资源管理。

构成自组网络一部分的移动电话之类的设备通常内存、处理能力和存储空间都很有限。这样的设备在成为网络的一部分时可以使用不同的服务。使用服务的设备称做服务的客户。每个服务在安装之后都会消耗一些资源。如果设备一开始就安装好所有的服务，那么就会有很多开销，会不必要地消耗很多资源。但另一方面，有的设备可能会需要高确定性的行为，所以必须一开始就把服务安装好。

随着时间的推移，某个设备可能不再需要已获得的一些服务。除非设备显式地终止同服务提供者的关系并释放服务以及相关资源，否则的话还是会不必要地消耗不使用的资源。这会导致设备和提供者的性能下降，因为会发生交换或者其他内存管理活动。此外，这还会影响对其他设备的服务可用性。

从前面的需求可以推导出下面这些作用力，它们构成了自组网络计算领域的特征:

- 效率（Efficiency）。应当避免不必要地获取服务，因为获取它们可能代价昂贵。此外，应当尽量减少不使用的服务造成的系统负载。
- 简单性（Simplicity）。对客户用到的服务的管理应当很简单，客户应该能够可选地显式释放不再需要的服务。
- 可用性（Availability）。不被客户使用或者不再可用的资源应当尽可能快地释放，以便可以让它们被新的客户使用。
- 适应性（Adaptation）。对服务的使用频率应当影响服务的生命周期。服务的释放应当取决于可用内存和CPU负载之类的参数。
- 时效性（Actuality）。当新的服务可用之后，设备不应该再使用过期的服务。

- 透明性（Transparency）。解决方案应当对客户透明。方案带来的执行开销应当尽可能小，软件开发的复杂性也应该最小化。

这些作用力扩展了第1章提到的作用力。6.3节描述了如何使用资源管理模式语言来解决这些作用力。

6.3 解决方案

为了解决前述作用力，自组网络通常会实现的解决方案可以用资源管理模式语言来描述。

每个自组网络都包含如下关键组件：

- 服务（Service）。服务被发布以供自组网络中的客户发现和使用。
- 查找服务（Lookup service）。服务提供者使用查找服务来注册服务。客户使用查找服务来发现服务。
- 客户（Client）。客户（比如移动设备）是使用服务的实体。客户使用查找服务来发现服务。
- 服务提供者（Service provider）。服务提供者是通过查找服务注册服务（或者服务的一部分）并向客户提供服务的实体。

上面的4个组件负责将资源管理模式语言应用于自组网络计算领域后的主要交互。它们间的依赖关系如图6-2所示。

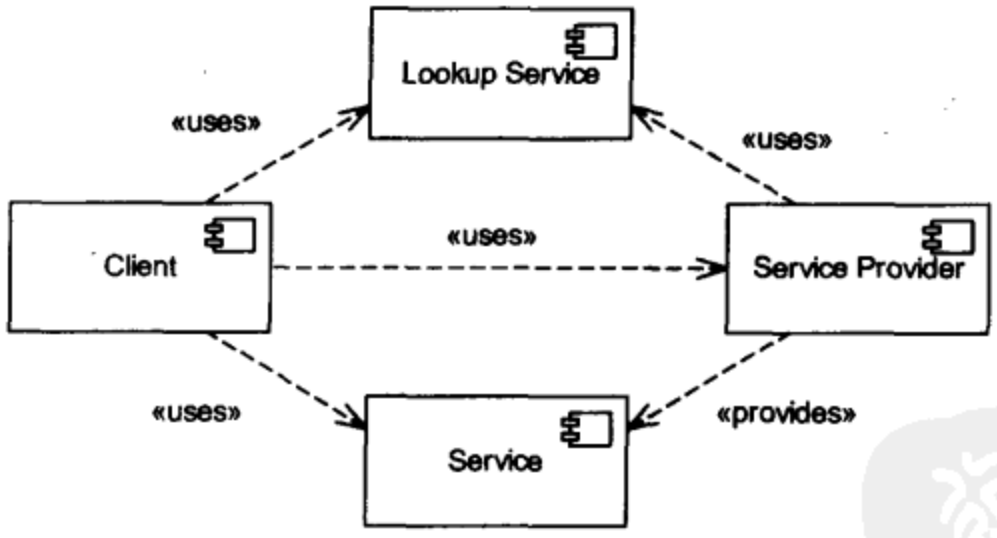


图 6-2

思考一下前面描述过的移动电话上的股票报价应用程序。报价程序包含如下部分：

- 作为服务提供者的证券交易所，运行于后端服务器。
- 作为服务的报价服务，运行于后端服务器。
- 移动电话的应用软件，作为客户。
- 作为前端软件的报价代理，在访问后端报价服务以及查询股价前需要下载到移动电话。
- 作为查找服务的服务目录。

图6-3展现了股票报价例子的部署。

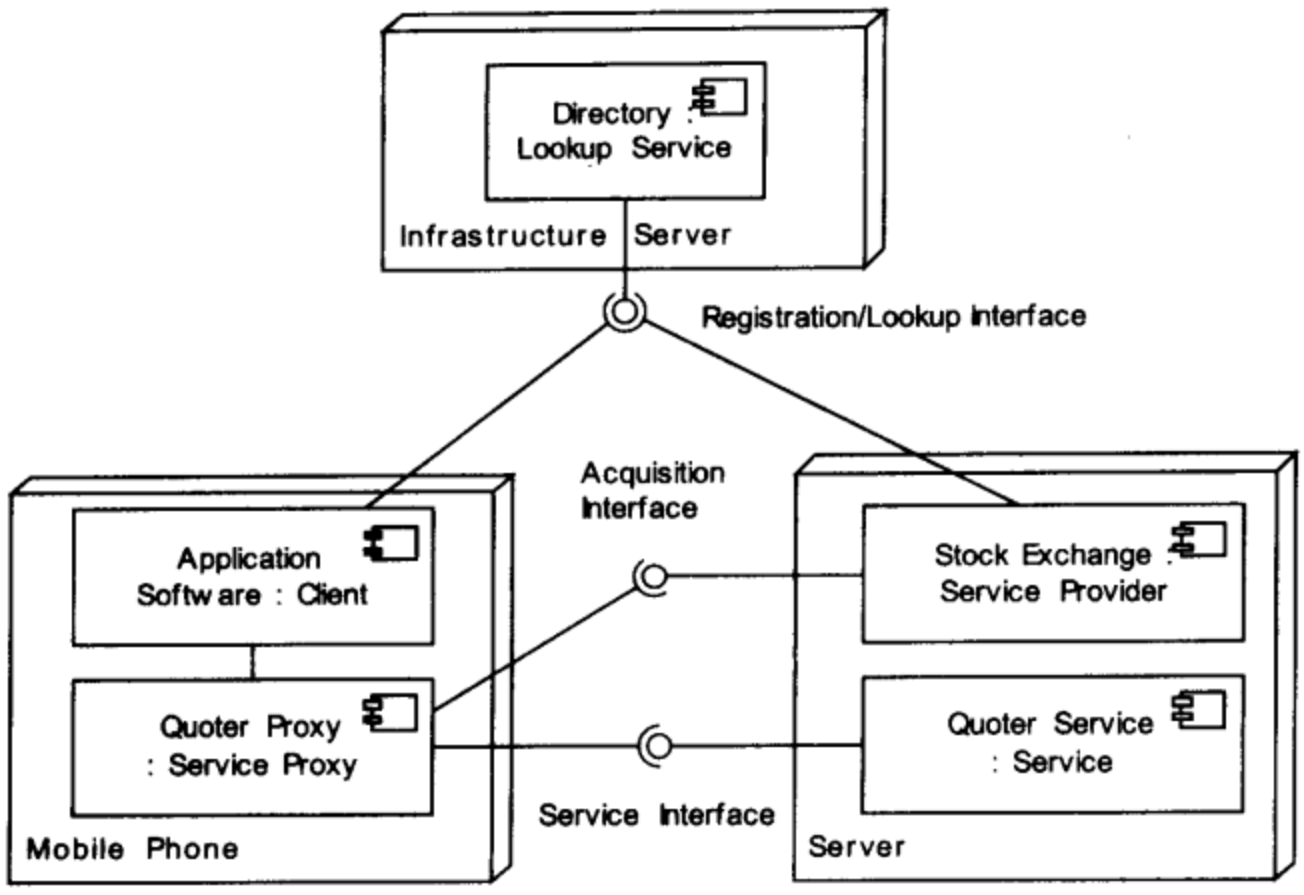


图 6-3

一开始，证券交易所创建股票报价服务并将其注册到目录，顺序图如图6-4所示。

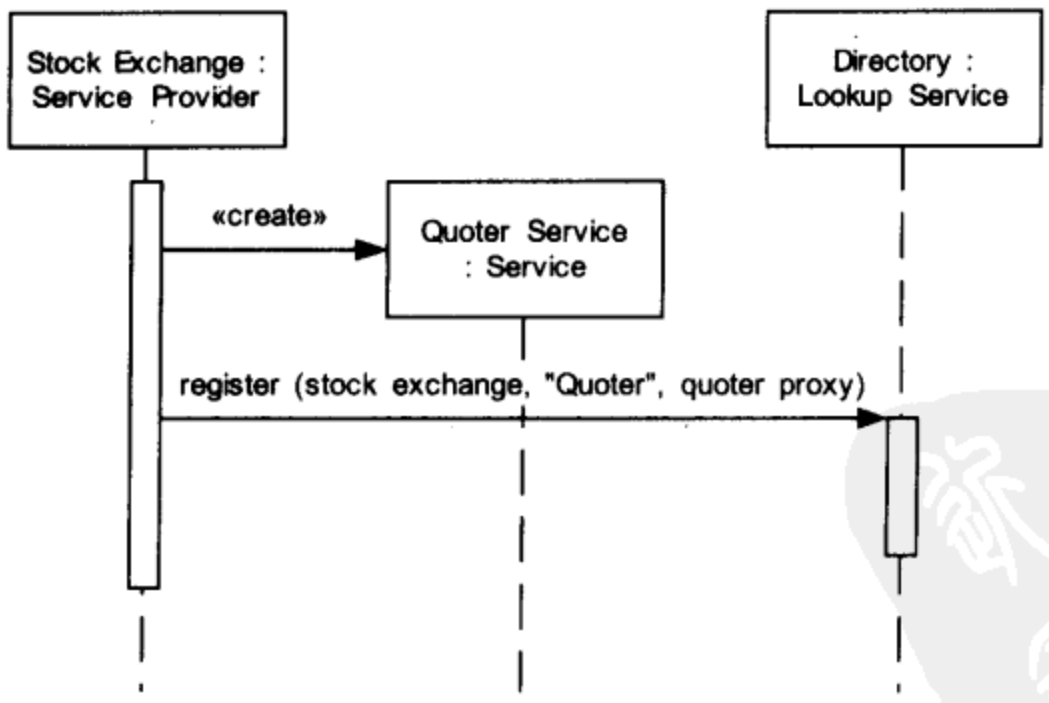


图 6-4

在注册过程中，证券交易所服务提供者会提交指向自身的引用以及一组描述它提供的服务的属性。此外，证券交易所服务提供者还会向查找服务注册报价代理。证券交易所服务提供者扮演的是资源提供者的角色，而报价服务则是它提供的实际资源。使用Leasing模式来自动从服务提供者的目录中删除不再提供任何服务的服务提供者（它们的租期已到期）。

为了优化移动电话中的资源使用，应用软件通常直到实际需要时才会获取和使用报价服务。

Lazy Acquisition模式描述了这种即时方式，它有助于为客户节省宝贵的资源。

当电话使用者请求报价服务时，移动电话的应用软件会查询目录，目录实现了Lookup模式。目录提供了到报价代理的引用，以及到对应于报价代理的报价服务所在的服务提供者的引用。报价代理需要证券交易所的引用，这样才能同报价服务的实现通信，报价服务位于证券交易所资源提供者中。因此，在安装了报价代理之后，应用软件向报价代理提供了证券交易所服务提供者的引用。报价代理使用这个引用来访问证券交易所的服务。报价代理成为应用软件的一部分，应用软件扮演的是客户的角色。下面的顺序图展示了这些步骤（见图6-5）。

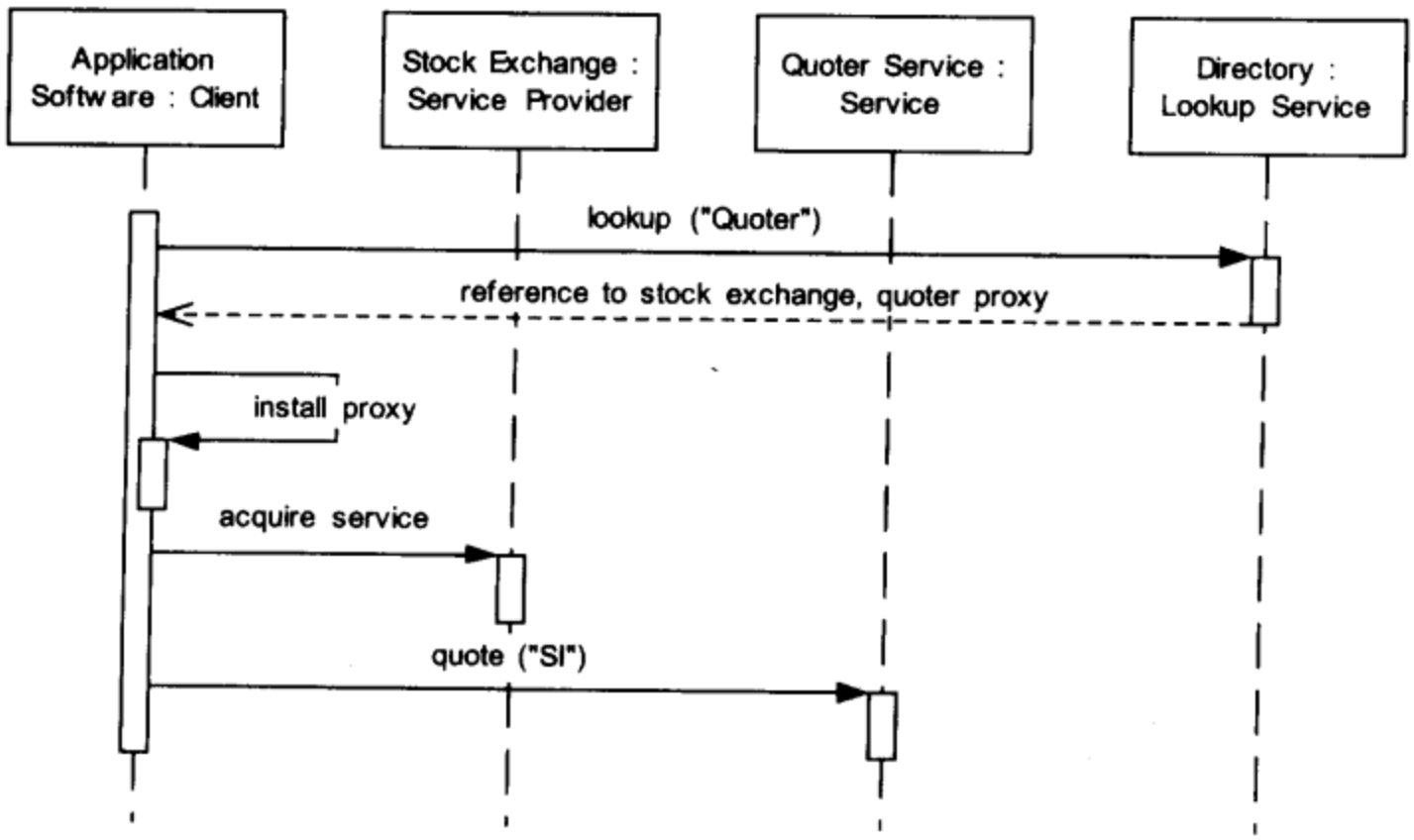


图 6-5

在通过目录找到服务之前，移动电话的应用软件必须先靠自己找到目录。Lookup模式详细讲述了如何找到目录、注册服务以及查找服务。对于这个场景，比较特殊的一点是会从查找服务下载客户代理。

一旦移动电话找到了服务，它就通过租借来获得对服务的访问。租借是在一定时间内保证对其的访问。每次租借都由客户和服务提供者达成一致。Leasing模式描述了这一点，以及其他可以支持的变体。

一旦租借成功，移动电话就直接使用服务。这通常会要求把服务或者服务的一部分加载进电话的地址空间。为了优化资源使用，这通常会尽可能晚地做，直到需要时才做。Lazy Acquisition模式描述了如何做到这一点。

当服务的租期已满，或者不再需要服务时，那么移动电话会停止使用服务，客户同服务以及相关资源的关联就会被清除。Evictor模式描述了这种资源管理方式。下面的顺序图描述了租期已满、关联清除的场景（见图6-6）。

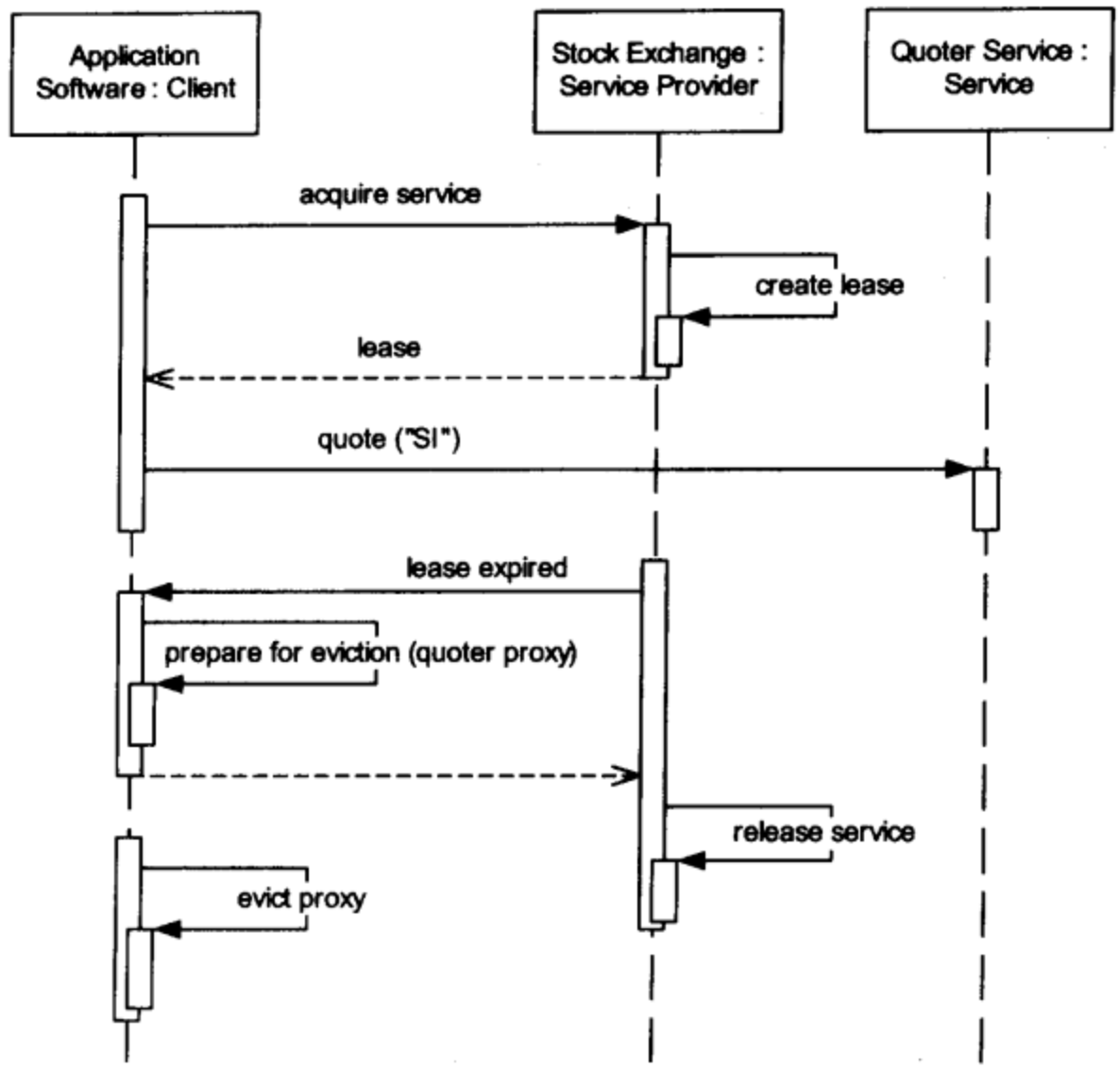


图 6-6

移动电话的应用软件首先获得服务，然后通过报价代理来访问服务、进行查询。如果没有定期使用报价服务，那么证券交易所（服务提供者）的租期就会结束，应用软件会得到通知。报价代理通常会注册到Evictor。当报价代理在一个预定长度的时间内未被使用时，那么Evictor就会选中它并将其从内存中清除。这个时候，证券交易所会自动释放这个服务。

回到对自组网络计算的更泛化的视图，图6-7展示了参与者以及它们如何同模式语言中的模式相互关联。

图6-7表明，客户使用了Evictor模式来移除服务。当客户获取服务时，使用了Lazy Acquisition模式。服务提供者首先使用Lookup模式来把资源注册到查找服务，然后客户使用Lookup模式来从查找服务中找到服务。最后，服务提供者使用了Leasing模式来处理查找服务的注册，客户也使用了Leasing模式来从服务提供者那里租借服务。

除了本书中描述的资源管理模式之外，其他的模式对于处理自组网络系统的需求也很有用。例如，客户可以用Pick & Verify模式[Wisc00]来随机地从一个服务池中挑选服务，以处理无法预先判断哪些服务可用的情形。类似地，如果服务正在被移动，所以指向它们的引用也随之变化，那么Locate & Track模式[Cohe00]也有助于跟踪位置的改变。

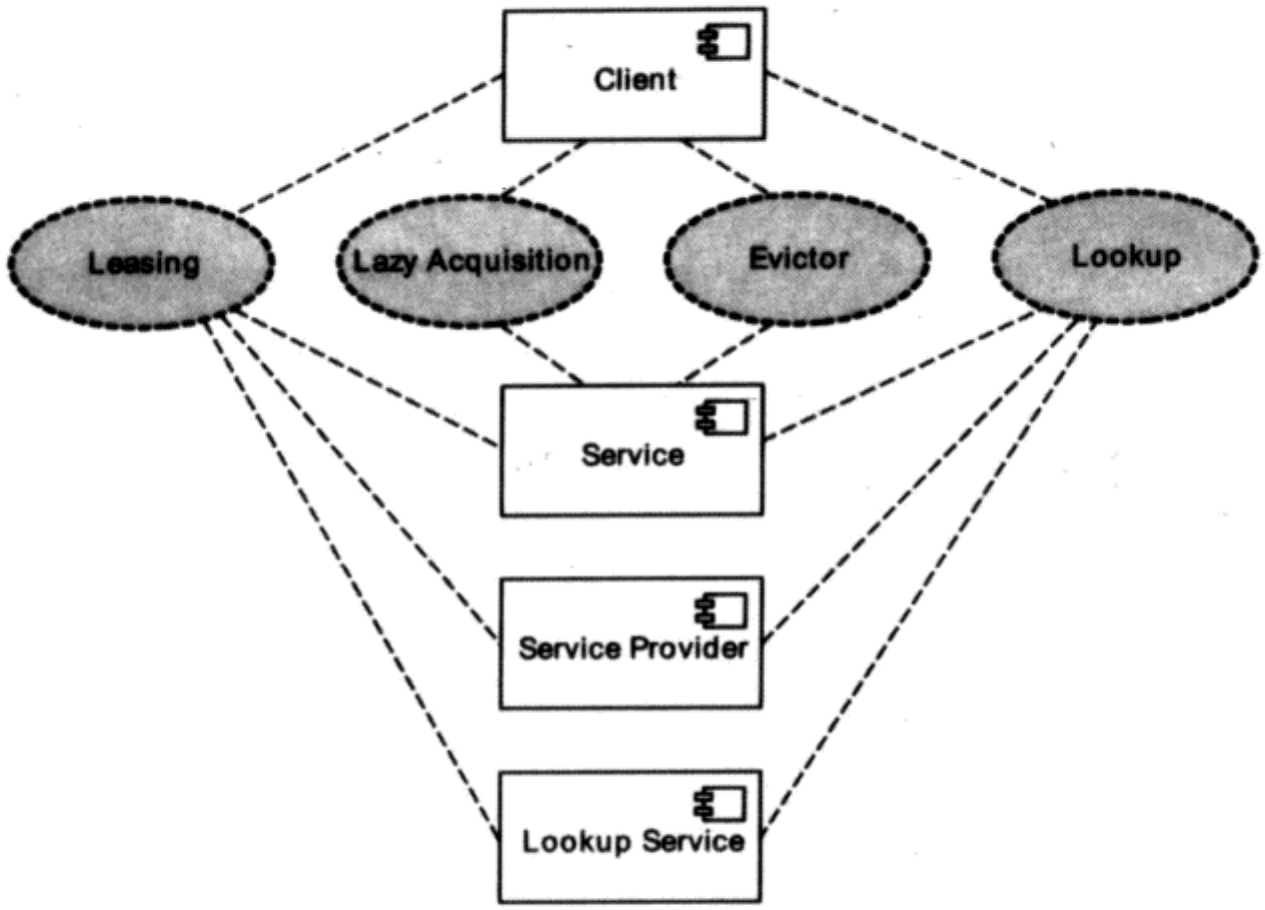

Client
Leasing
Lazy Acquisition
Evictor
Lookup
Service
Service Provider
Lookup Service

图 6-7

第 7 章
案例分析——移动网络

“我总是希望我的电脑可以像电话一样容易使用。我的愿望实现了，现在我不知道如何使用我的电话了。”

——Bjarne Stroustrup

重点

本章描述一个特定领域的例子，展示第2章、第3章和第4章中展示的模式如何应用于一个领域的资源管理需求。描述的例子是用于移动电话的电信系统。如第5章中所描述的那样，我们遵循这些步骤，首先找出系统中的热点。因为在热点和必须解决的作用力之间有紧密的联系，我们直接将资源管理模式语言用于解决作用力。

本章首先展示了该领域中一些关键概念的概览，然后详细讨论并试图确认这个领域中特定的需要解决的作用力。使用前面章节中展示的模式，我们解决了这些作用力并使用资源管理模式语言描述了一个解决方案。

7.1 概览

移动网络包含了从移动电话接受呼叫并把呼叫转到核心电信网络所需要的所有网络元素。

所涉及的网络元素之间职责的部署协议是由Universal Mobile Telecommunications System（UMTS）标准[3GPP04]定义的，有时也称做“第三代”（3G）移动通信系统。UMTS预期会在几年内替代现有的技术，比如欧洲的Global System for Mobile Communications（GSM）。我们在本章中描述的构架来自为UMTS开发软件的真实项目。

移动网络通常包含3种不同的网络元素[LoLa01]：基站（也称做NodeB）、射频网络控制器，以及操作和维护中心（OMC）。此外，还有一些其他的网络元素为网络的正常工作扮演了重要角色，我们将其作为核心网络的一部分加以描述。但是，对于这个案例讨论的目的，我们主要关注3个主要的网络元素，那就是基站、射频网络控制器以及操作和维护中心。这3个网络元素的职责用图7-1描述。

- 基站。基站负责同移动电话通信，有一个或多个天线同它们相关联。天线用来通过射频信号发送数字化的音频。基站把呼叫从移动电话转移给射频网络控制器，如果呼叫接受者是移动电话的话，控制器再将其转移给另一个基站；如果呼叫接受者是固话（固定电话）就转移到核心网络。基站由操作和维护中心操作和维护，操作和维护中心会控制网络中几乎所有硬件和软件元素的配置。

 基站由嵌入式计算机组成，分布在广大的物理区域。它们必须是高度可靠的，以便尽量减少修理的需要。

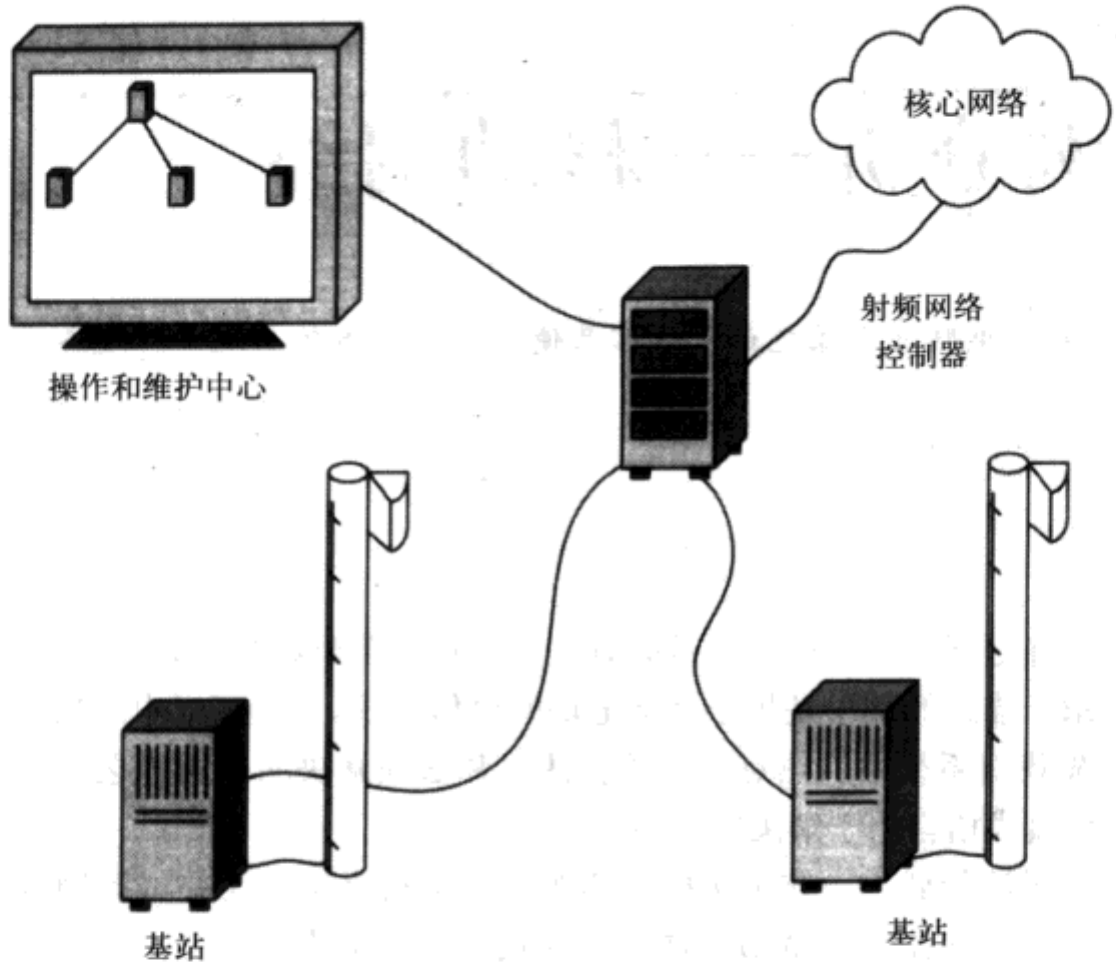


图 7-1

- 射频网络控制器（RNC）。RNC负责基站、其他RNC以及核心网络之间的呼叫中转，这取决于呼出者和接受者分别位于何处。为此，它同其他网络元素中的呼叫处理组件通信。RNC由操作和维护中心操作和维护，这同基站一样。

 RNC是高性能的企业系统，运行于通用操作系统。因为它们是移动网络的中心组件，所以它们必须高度可靠。

- 操作和维护中心（OMC）。OMC管理RNC和基站的网络。RNC是通过直接连接进行操作和维护的，到基站的连接则通过RNC路由。OMC负责监视网络元素的正常工作，并且允许操作员在必要的情况下干涉。除了对基站和RNC的本地维护终端，OMC是移动网络中仅有的有用户界面的网络元素。通常情况下，OMC实现为分布在多个节点的多个进程上的软件系统。这包含了负责管理网络拓扑的子系统，负责保存关于拓扑的数据库信息的子系统，以及负责处理OMC的用户界面并同网络元素交互的子系统。OMC负责网络拓扑上的所有操作，比如重新安排和优化。

这3类网络元素紧密合作。基站同所有移动电话都保持联络，这是通过有规律地通过公共频道交换信息而做到的，这样双方就都为呼叫建立做好了准备。

当移动电话的用户拨打号码时，移动电话通过到基站的公共频道要求向网络建立呼叫。移动电话发送包含被呼叫者电话号码以及其他额外信息的数据包。

此时，RNC决定是否应该同核心网络建立连接。如果被呼叫方是固话，或者只能通过核心网络抵达，那么就同核心网络建立连接。如果被呼叫方是同一个RNC通过另一个基站可抵达的移动电话，那么RNC就决定把呼叫转发给另一个基站。在同核心网络核对过（为了收费和验证的需要）之后，RCN建立连接。图7-2描述了建立到固话的呼叫的场景。

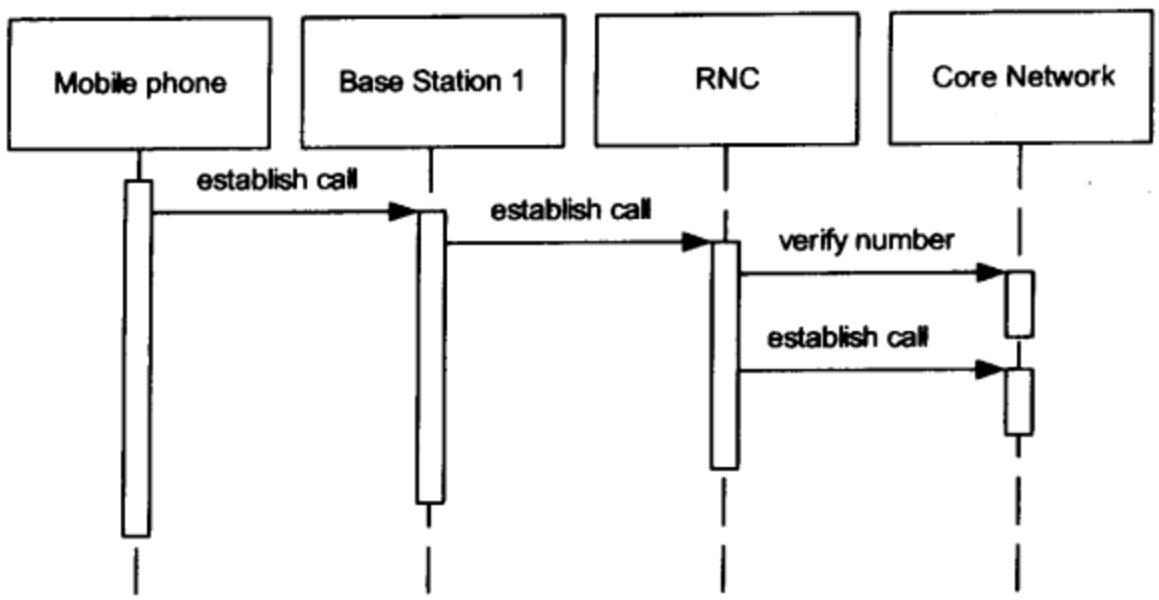


图 7-2

当基站检测到移动电话的射频信号过于微弱时，它就选择另一个可以从该移动电话收到更强信号的基站。在一次事务中，第一个基站把语音连接转交给第二个基站，以确保RNC和移动电话之间的连接不丢失。从一个基站到另一个基站的转交对于移动电话用户来说是如此透明，以至于用户不会注意到转交的发生（哪怕处于以200km/h前进的火车中）。下面的顺序图描述了呼叫转交的场景（见图7-3）。

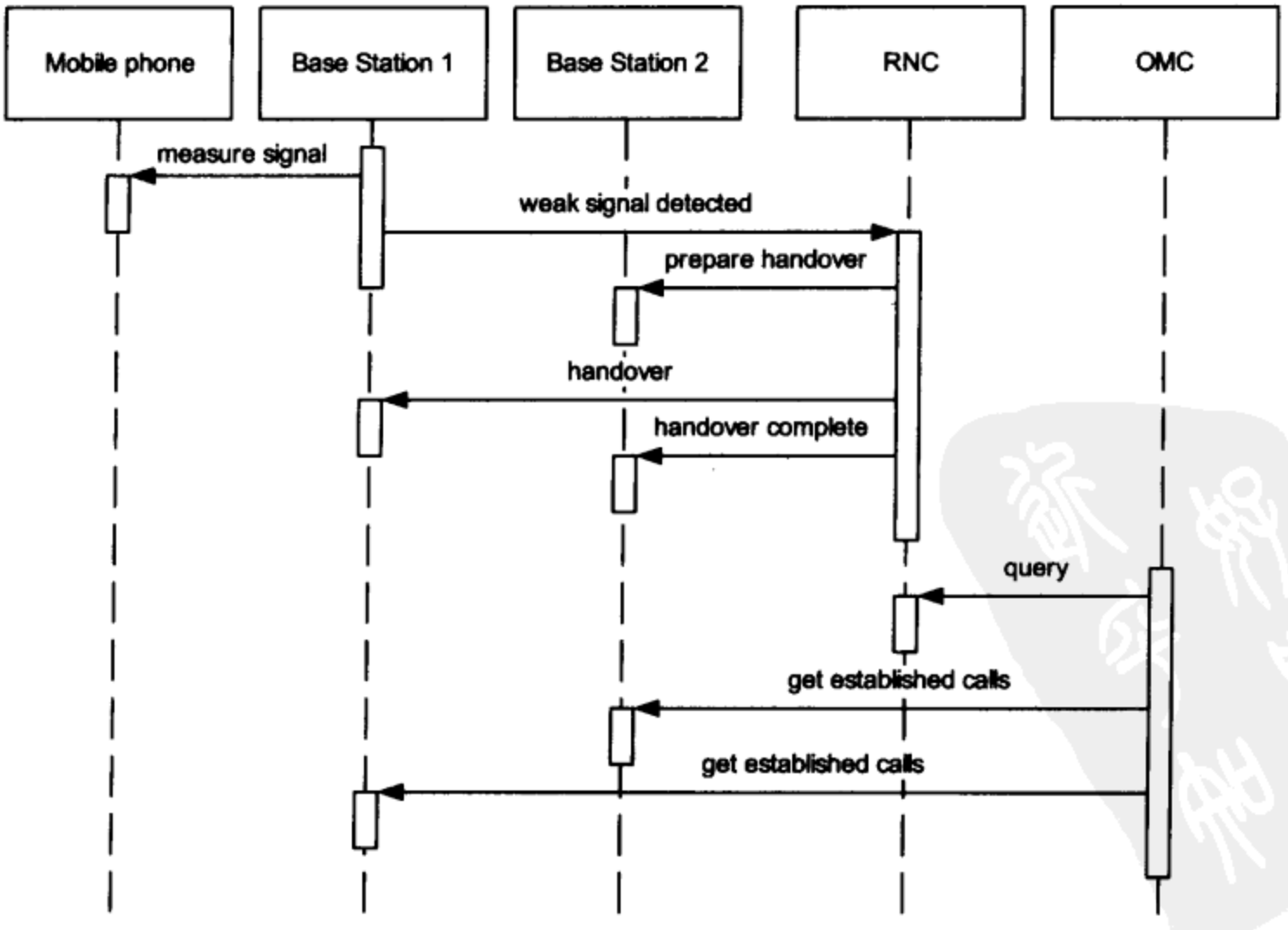


图 7-3

基站和RNC都必须设计为可以确保呼叫建立和转交的端到端的服务质量，哪怕有多个用户同时在用也应如此。

7.2 动机

因为移动网络的3种类型的网络元素执行非常不同的工作，所以它们具有不同的硬件和软件构架。结果就是，不同网络元素的需求相当不同。但是，它们依然面临着很多共同的作用力。

- 可伸缩性。设计和实现移动网络的一个主要难点就是处理几十个（如果不是成百个的话）网络元素。因为每个网络元素都需要在软件中表示出来，所以我们需要很复杂的资源管理方法。移动网络必须可以伸缩，以支持额外的网络元素和额外的天线，以及额外的移动电话。结果就是，高效的资源管理对于处理必要的额外通信和协调非常重要。
- 可靠性。移动网络中的系统必须高度可靠，因为故障可能会影响到大量客户的移动服务，对服务提供者可能会造成严重的财务和非财务影响。

 另外，如果网络元素因为软件或者硬件故障而不再可访问，服务技术人员在野外修理问题的代价可能会很昂贵。
- 服务质量（QoS）。移动网络系统必须遵守一些QoS限制，包括精确定义和可预测的启动时间、对新的连接请求的响应时间，以及基站之间语音连接的无缝转交。为了确保满足这些限制，在软件构架的所有层次都需要考虑它们。
- 性能。移动网络必须满足严格的性能要求，这包含每秒可以处理的呼叫建立的最小数目，以及基站间每秒可以进行的转交的最少数目。此外，移动网络中的每个系统通常都会被要求同时处理一定数目的呼叫。移动网络中的单个元素也应该具有较短且固定的启动实现。系统的响应时间必须短，这样服务使用者才会满意。
- 分布性。移动网络中的操作和维护软件分布在多台机器中。OMC对所有网络元素执行一般的监管，而基站及RNC中的本地操作和维护（O&M）单元则监控它们的内部状态，只把过滤过的故障信息发送给OMC。整体系统的复杂性来自由地理决定的天生的分布性。
- 故障处理。移动网络必须处理很多不同类型的错误和故障。基站和RNC中的故障，不管是来自硬件、软件还是连接，都应该记录下来并发送给OMC。如果基站因为本地故障同OMC失去了联系，它通常会重启，从而可以再次被监控。因为RNC提供的服务必须是不间断的，所以RNC必须以不同的方式处理故障。RNC通常通过冗余机制来避免硬件故障。

7.3 解决方案

本节描述如何用资源管理模式语言来解决上述的作用力。解决方案描述了基站和OMC的构架，并展示了资源管理模式语言如何可以用于解决这些组件的功能和非功能需求。就功能分布于多块CPU板以及O&M和呼叫处理的分离而言，RNC构架同基站类似。同基站类似，它也是由OMC控制。这导致对模式的使用也类似。前面指出的作用力在解释网络元素的构架时都会一一提到。

这里描述的例子是特定于电信领域的，但很重要的一点是要理解这些模式和模式语言是独

立于任何领域的。它们可以有效地应用于很多不同的领域以解决这些领域共同的作用力。目标是采用“面向问题”的方法，使用模式和模式语言来解决一组问题。

基站构架

基站的构架分成两个关键功能单元：呼叫处理（CP）单元和操作维护（O&M）单元（见图7-4）。

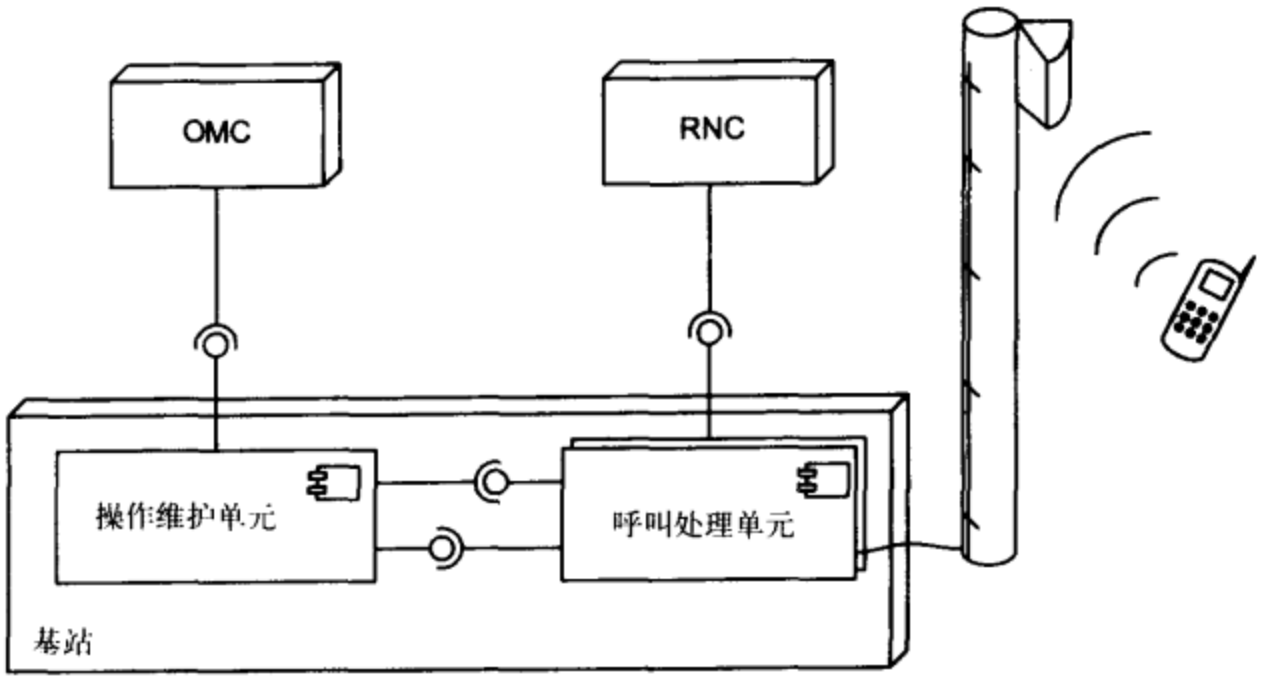


图 7-4

呼叫处理单元

基站的呼叫处理[Hanm01]单元通常会分布在多块CPU板上。这是因为单个CPU的性能不足以处理大量的呼叫。多块CPU板也提供了冗余，确保了单个硬件故障不会导致整个基站的损失。一块处于中心但冗余的CPU板负责协调其他CPU板，其余CPU板则负责实际的射频连接。在物理上，每块CPU版都通过内部通信网络连接到其他所有CPU板。

呼叫处理包含下列职责：

- 语音连接管理。基站表现为在移动电话和RNC之间必须建立的连接的中转者。
- 内部连接管理。内部的CPU板必须相互通信以提供语音连接。需要做到这一点。必须在CPU板之间建立连接，并避免资源冲突。
- 质量监控。如果来自一个移动电话的通信信号变得过于微弱，基站必须通知RNC，这样RNC可以寻找一个具有更强信号的基站。

简而言之，呼叫处理包含了与管理移动电话和核心网络间的连接建立、转交和连接终止相关的所有事情。

操作维护单元

基站的操作维护（O&M）[Hanm02]单元用一棵“被管理的对象”（受管对象）的树来表示所有硬件元素以及部分软件组件。被管理的对象可以是一块集成芯片，也可以是整块CPU板。

本地O&M单元负责监控和配置所有的硬件和软件元素，这样呼叫处理单元就可以专注于它的任务而不必分心处理资源管理问题。下面的图表展示了简化的受管对象树，它包含层次组织的受管对象（见图7-5）。

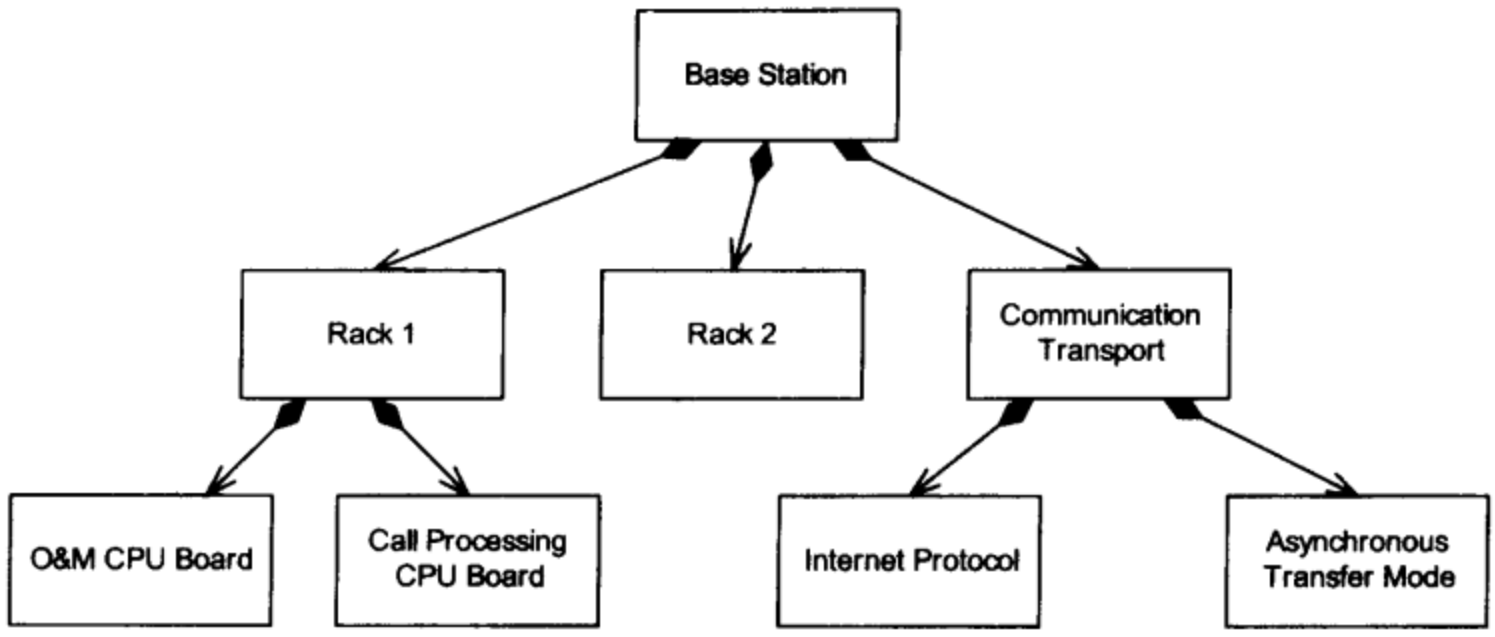


图 7-5

O&M单元的逻辑分成本地部分（由基站实现）和远程部分（由OMC实现）。OMC可以远程访问受管对象的树形结构（见图7-5）。这使得OMC可以对基站执行O&M操作，控制受管对象相对应的（硬件）部件。

基站的O&M单元将其接口注册到OMC接受管理并报告故障。通过OMC抵达O&M接口的命令主要是由专门用于O&M的CPU板处理的。

值得指出的是，基站可以独立于OMC而工作，所以它不需要永远都保持连接状态。

基站功能规约

本小节描述基站构架中的资源热点。资源管理模式语言是处理这些热点的有用参考，因为它解决了很多功能和非功能的需求。

灵活的应用程序发现

为了简化运行于基站的应用程序的配置和管理，它们被实现为组件。这意味着每个应用程序自身是内聚的，具有精确定义的服务接口，并且是可重用的。服务接口描述了单个应用程序交互的语法和语义。

应用程序组件通常包含操作和维护、呼叫处理、连接管理和启动管理。

为了增加可靠性，运行于CPU板上的应用程序以多个进程来运行。当启动时，应用程序通过它们的进程间通信机制互相连接。为了互相连接，应用程序如何找到彼此的服务接口（见图7-6）？

每个服务接口都同一个引用相关联，以便应用程序间的灵活和迟绑定。引用独一无二地标识了组件的特定接口。既然应用程序不是静态连接的，那么它们后来如何连接呢？

Lookup模式可以帮助以优雅的方式解决这些问题。每个应用程序都把自己的服务接口注册

到一个查找服务。而且每个应用程序都会查找它需要连接的服务接口的引用。因此，每个应用程序都扮演了服务提供者的角色，服务接口就是应用程序提供的资源。

请注意，查找服务本身是一个组件，需要在被访问之前就建立好。为了解决这个问题，查找服务通常是静态配置的。

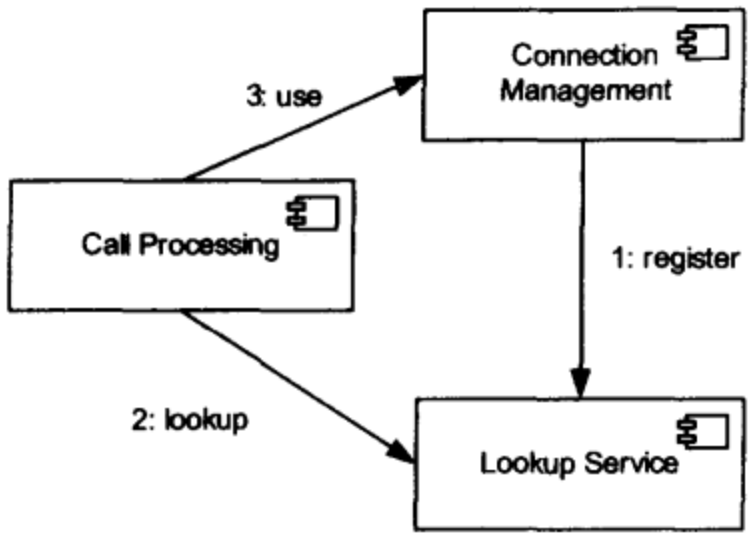


图 7-6

连接管理

因为从基站到OMC的连接既不可靠也不固定，因为也可以通过连接到同一O&M接口的本地维护终端（比如一台笔记本电脑）来进行维护，所以会出现很多损坏和丢失的连接，这就浪费了有价值的资源。

Evictor模式解决了这个问题。使用一个Evictor组件的实例，所有的连接都注册到这个实例。Evictor组件负责主动或者被动地校验以循环利用未被使用的组件，这样资源消费就总是被最小化。减少资源消费很重要，因为服务提供者不能承受把技术人员派到野外去复位资源耗尽的基站的开销。

高性能

当新的语言通道需要建立时，基站必须满足高性能的要求。因为连接请求可能涉及多块CPU板，这些CPU板必须互相协调，这可能涉及数据复制以及在跟踪连接ID和物理传送资源的关联的容器中分配新的条目。动态内存分配和同步属于计算机系统中最昂贵的操作。必须避免在语音连接建立的关键路径上执行昂贵的操作。

为了避免在性能至关重要的运行时执行昂贵的资源获取操作（比如内存分配），使用了Eager Acquisition模式。Eager Acquisition模式建议把昂贵的资源获取操作放到程序执行的较早阶段。应用程序启动时一般性能不会那么关键，虽然也必须足够快，但多花几微秒或者几毫秒不会有什么关系。但另一方面，在建立呼叫的时候这样的延迟可能就是无法接受的。

基站的操作和维护主要是由一块专用的CPU板来执行的，但是所有CPU板的硬件和软件元素都必须在OMC中表示出来（见图7-7）。专用的CPU板通常会远程地调用对其他CPU板的维护和状态操作。状态信息包含了使用中的连接数目、传输失败的数目或者特定操作的失败率之类的属性。如果需要执行很多这样的状态操作，它们可能会造成长的延迟。如何降低访问远程

CPU板上的属性所需的时间?

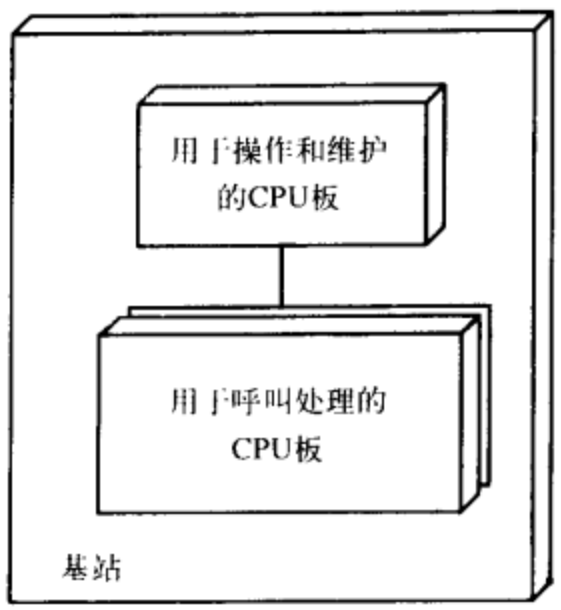


图 7-7

Caching模式允许CPU板之间的执行更好地解耦合。专门的CPU板所需的来自其他CPU板的数据被暂时保留在缓存中。对系统分析和优化而言，对另一块CPU板的信息的立刻访问特别重要。一个风险是不注意地使用了过时的信息，所以当CPU板上的数据变化时必须使用特殊的协议来避免这个风险。

故障处理

基站和OMC之间的通信是由OMC建立的，后面的“OMG构架”一节描述了这一点。关于OMC中的软件更新，当有新版本的服务接口可提供时或者当重组时新的OMC实例关联到基站，那么到原来的OMC的服务接口可能会变得无效。基站如何确认到服务接口的引用是否不再有效或者未被使用?

我们使用Leasing模式来把租约同OMC服务接口的引用相关联。由OMC提供这一租约，基站定期地保证其处于活跃状态。只要可用，OMC就可以延长租期。如果OMC变得不可用，那么基站所维护的引用就变得无效。在这种情况下，因为OMC不会更新租约，所以基站会把引用标记为无效，并不再使用这个到OMC的引用。基站会等待，直到联系到另一个OMC实例，创建了新的租约。OMC联系基站的协议在“站点发现和通信”一节中描述。呼叫处理不受OMC切换的影响。

Leasing模式确保了故障或者改变会自动且自治地进行。资源（比如到服务接口的引用）会被及时释放。

并行

OMC会在基站执行几种动作，比如测量某个特定语音通信通道的性能（见图7-8）。对此，基站中的O&M组件通常会建立一个新的线程。因为这一动作可能会花较长的时间，必须有其他线程可用以保证基站的响应性。如果OMC执行多个需要长时间的动作（如果需要执行多种测量，那么很可能会发生），那么就会有几个线程建立并且并行运行。这可能会导致创建线程的开销和高的系统复杂，从而可能会影响到呼叫处理单元。

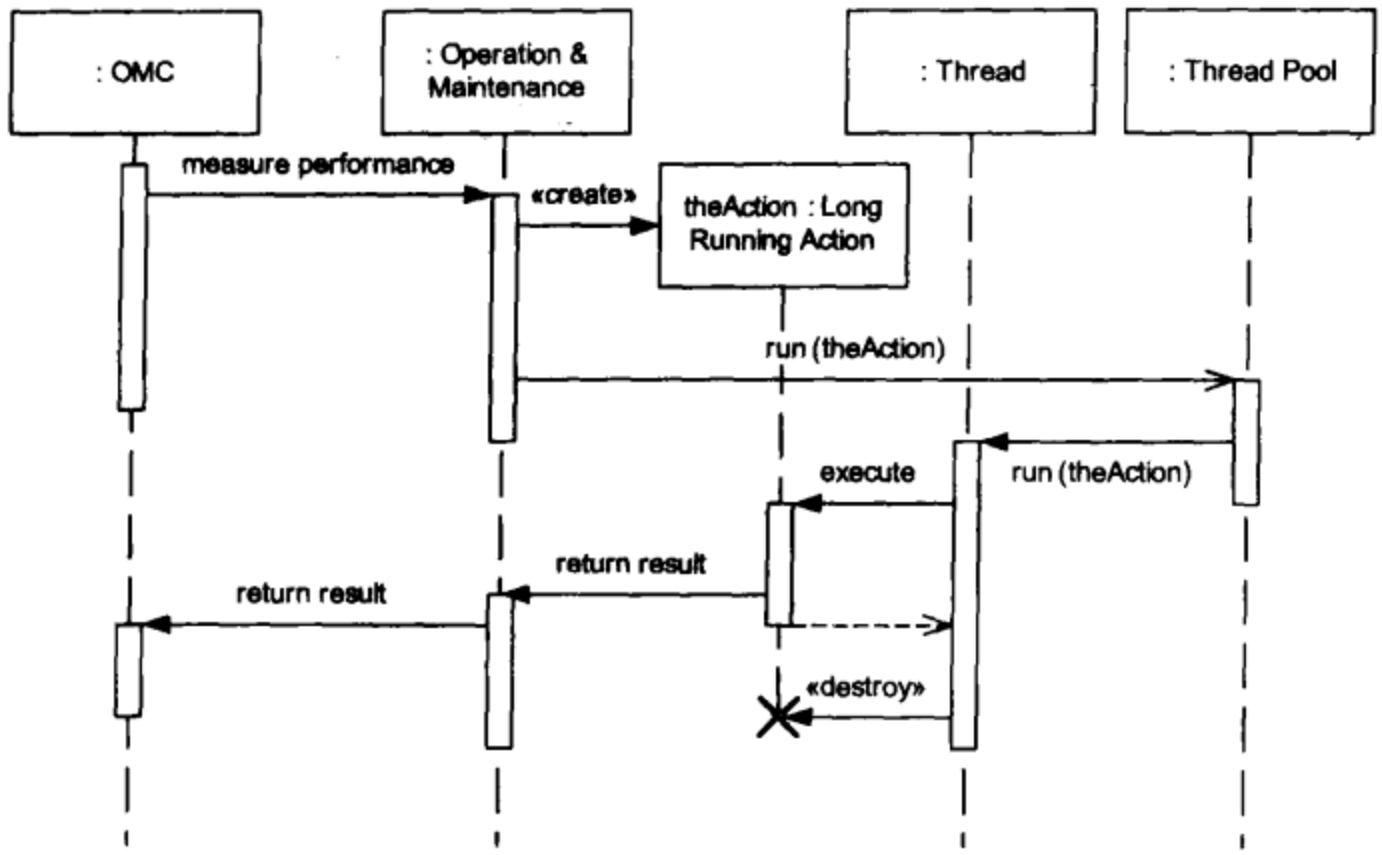


图 7-8

为了避免这样的开销，并避免出现负载过度的情形，我们使用了Pooling模式。线程池管理用于执行长时间动作的线程。线程池配置了可以同时运行的线程的最大数目。O&M组件把长时间动作的执行委托给线程池，而不是动作触发时就创建一个新的线程。线程池内部会提供一个线程来执行动作。如果达到了线程的最大数目，而又有新的动作触发，那么动作会被封装成一个命令（如Command Processor模式[POSA1]所描述的），并且为稍后的处理而排队。

小结

图7-9把软件组件同时用的模式联系了起来。

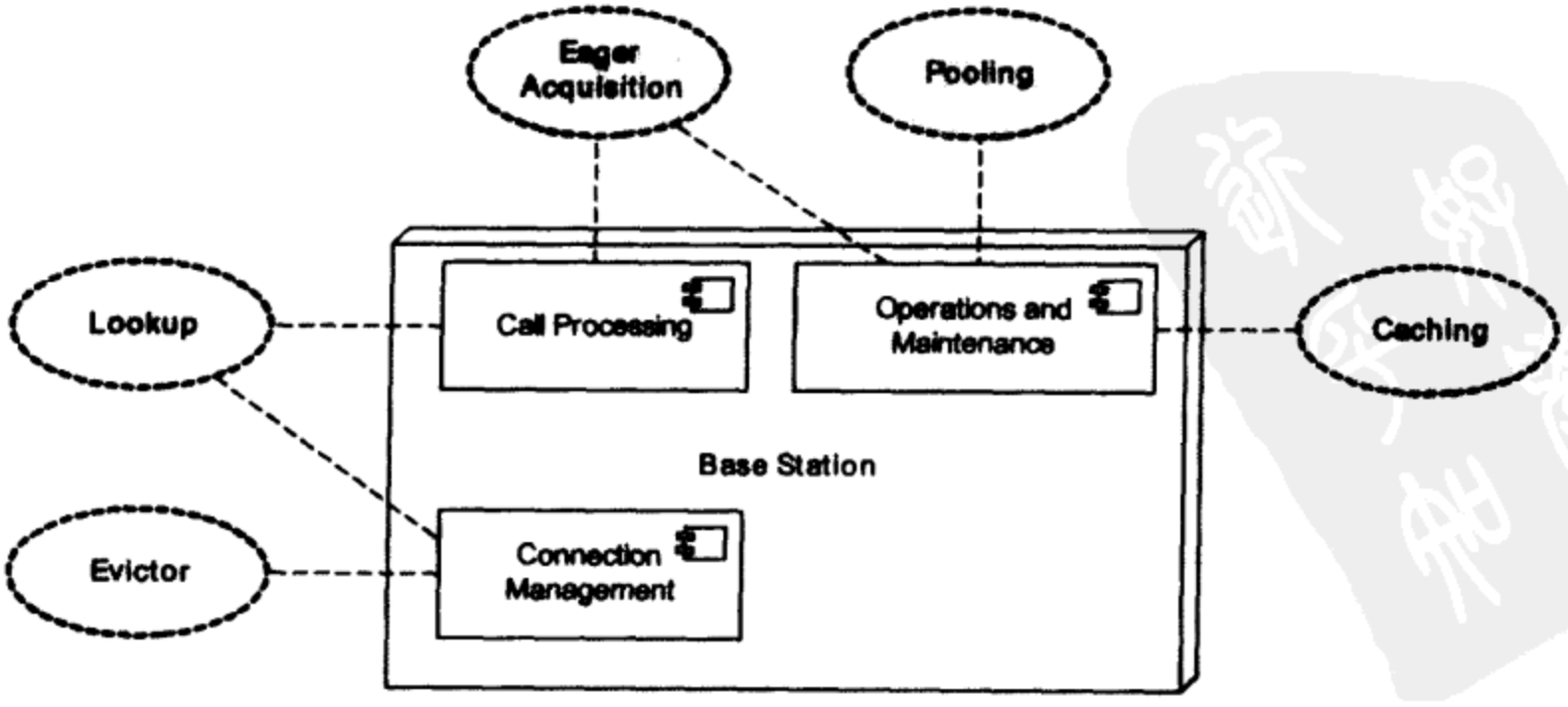


图 7-9

OMC构架

操作和维护中心（OMC）是一个网络管理系统，负责管理和监视移动网络元素（比如基站和RNC）。它类似于一般网络元素（包含路由器和网络交换机）的管理系统。

为了可以管理移动网络，OMC以拓扑树的形式提供了网络及其元素的视觉视图（见图7-10）。这棵树包含了所有网络元素的受管对象树。图形界面比命令行界面（command-line interface，CLI）更直观。但是，图形化地显示几十个（如果不是上百个的话）元素的状态，并且提供同它们的快速交互，这要求OMC在内存中保留关于网络元素的大多数信息。这个图形界面直接访问这一信息。

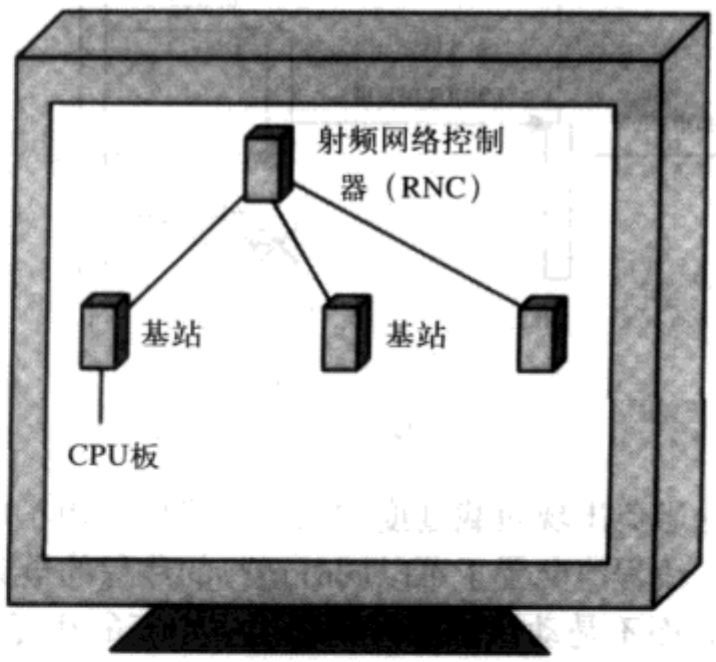


图 7-10

访问网络元素以获取它们的实时状态信息可能会很昂贵，因为这会引入长的延迟。并且，如果需要在OMC启动时获取这一信息，那么在图形用户界面可以显示足够信息之前需要等很长时间。为了避免网络延迟以及获取网络元素状态的处理时间，OMC把所有网络元素信息都保存在数据库中。在启动时，受管对象的内存表示从数据库中建立，稍后会基于网络元素实际的物理状态进行更新。网络元素的状态改变会先被更新到内存。如果状态改变是关键性的，它们会被立刻持久化，否则的话会在后台持久化。

不可能把所有网络元素的状态都保存在内存中，因为大型网络中会有大量的数据。而且，OMC从网络元素收集状态信息以及处理状态信息的功能是分布在多个节点中的，它们通过数据库交换信息。

OMC通常监视的网络元素数目会有好几千。每个网络元素都拥有很多属性，反映其硬件状态（比如CPU板和射频天线）以及软件元素（比如建立的连接和可用的内存）。很容易看出，把所有这些信息都持久化保存需要很大的数据库。此外，遍历拓扑树需要消耗很多处理能力。因此，通常OMC运行在具有多个CPU、大内存以及很多磁盘空间的企业级服务器上。硬件通常包含一个机器集群以支持负载分布和冗余。这会带来很多新的挑战，比如如何让集群中的所有机器的子状态保持一致（见图7-11）。

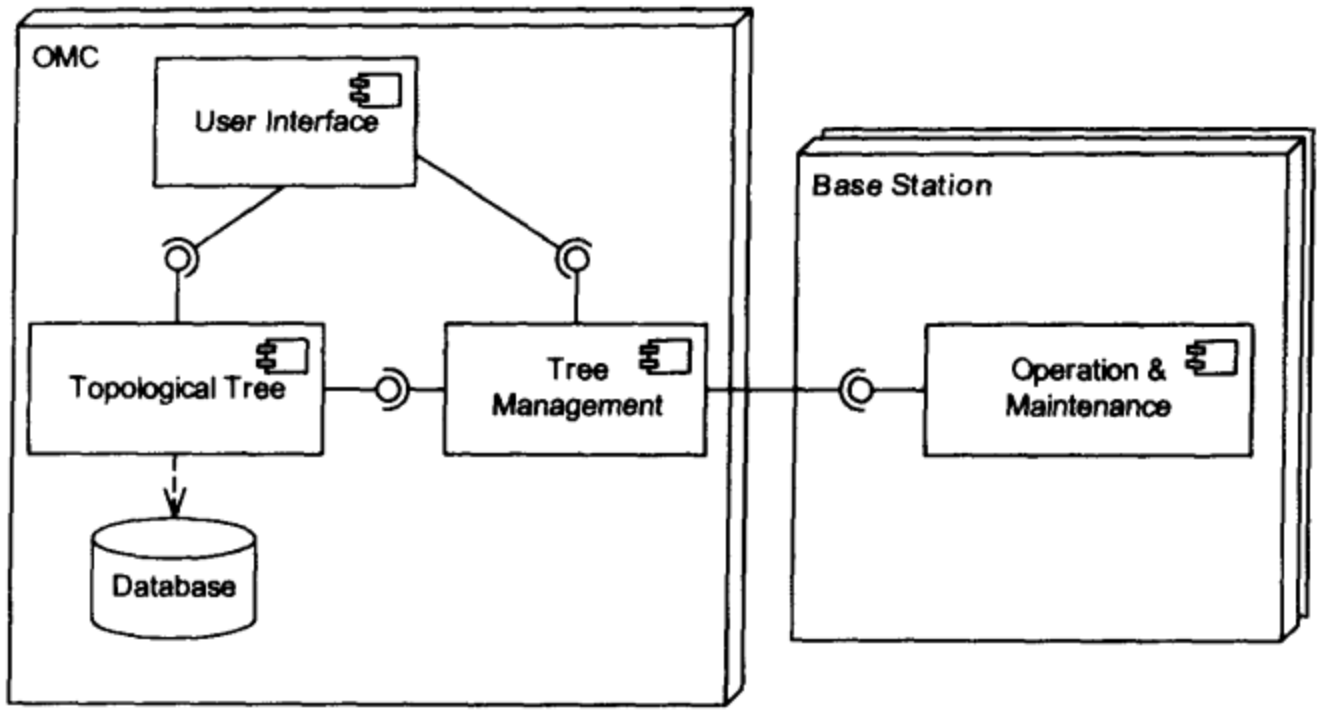


图 7-11

总体的OMC构架遵守经典的Model-View-Controller模式[POSA1]，并可以进一步分成如下关键部分：

- 用户界面，用来允许操作员和网络元素的交互（View）。
- 拓扑树，用来在缓存中存储同网络元素相关的信息（Model）。
- 树管理器，用来使得用户界面可以遍历网络元素（Controller）。

拓扑树完整地存在于数据库中，它的一部分存在于内存中。树管理器负责联系网络元素（比如基站和RNC）的O&M单元以获得实际数据。对于某些属性，比如需要发出警报和通知的情形，网络元素会主动地把信息推给OMC。

OMC功能规约

本小节描述OMC设计中的热点，以及资源管理模式语言如何通过解决很多OMC的功能和非功能需求而解决了这些热点。

站点发现和通信

OMC负责移动网络中的基站和RNC的操作和维护。为此，它会远程连接到基站和RNC。同基站通信的网络通常基于TCP/IP，由IP地址来区分基站。

因为基站是一个被动网络元素，所以它不负责寻找OMC以及找到它的IP地址。相反，OMC会配置好所有基站和RNC的IP地址。但是，知道网络元素的IP地址并不够，因为不知道O&M组件的具体服务接口。

为了解决这个问题，我们使用了Lookup模式来发现网络元素的O&M服务接口。OMC使用查找服务来记住网络元素的IP地址和那个基站的O&M服务接口之间的联系。对此，查找服务使用了特化的基于FTP（文件传输协议）的自举协议。查找服务使用IP地址来连接到网络元素，并恢复一个包含服务接口的文件，比如一个字符串化的CORBA[OMG04a]对象引用。然后，这个服

务接口被查找服务内部的表格所记住，并会传回给OMC的树管理器（见图7-12）。

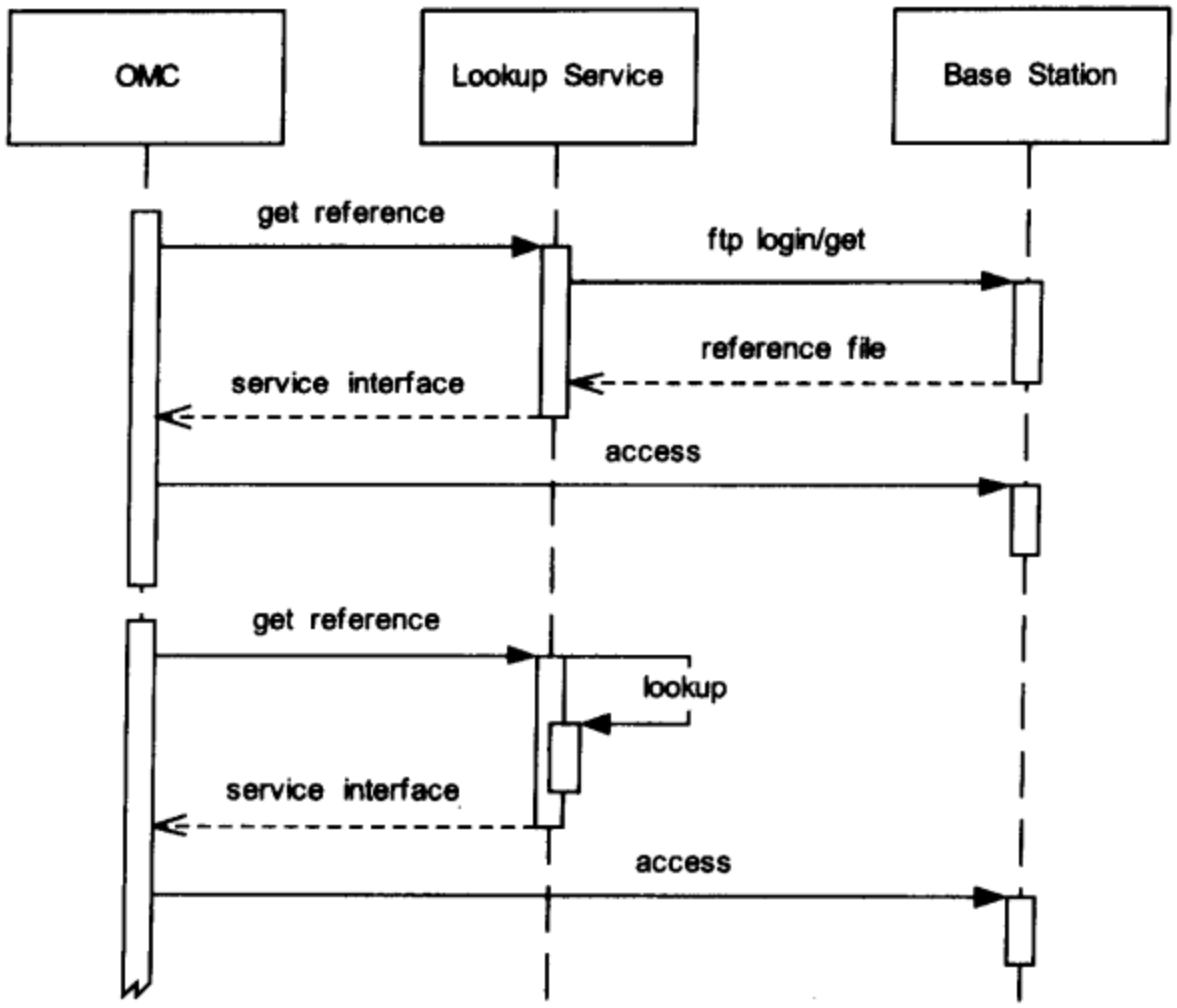


图 7-12

在这之后，OMC应用程序可以根据需要通过查找服务来查询服务接口。一开始通过FTP的自举只在服务接口变得无效的时候（比如因为站点软件的升级）才需要。因为这样的更新是由OMC自己进行，所以它知道何时触发查找服务来自举服务接口。

网络状态维护

OMC表示了网络元素以及它们在拓扑树中的连接。拓扑树是OMC的核心，在提供OMC的主要功能方面扮演了关键的角色。树包含了表示通过通信频道相连接的远程网络元素的节点。

因为网络连接天生就是不可靠的，所以单个网络元素的状态被缓存了，这样一旦忽然断开了连接，当前的状态还是可用的。使用Caching模式使得已经被获取的基站状态哪怕在到基站的连接不可用的情况下都可以被访问。为了使得通过缓存的访问尽可能透明，使用了检测基站可访问性的Cache Proxy[POSA1]。这个代理会在基站变得不可访问的情况下把状态请求自动转发到缓存。为了向操作员表明网络元素的状态来自缓存，网络元素通常会被突出显示，比如用一种不同的颜色，这样操作员就可以知道基站不可访问。操作员可以接着工作，但会意识到所显示的状态可能过期了。

高性能

Caching模式不仅有助于覆盖不可达的网络元素，还可以提升用户界面的性能。OMC需要可

以很快响应用户的输入（比如选择网络元素和查看它们的属性）。快速的响应时间很重要，因为操作员需要可以在紧急情况下快速部署弥补措施。如果网络元素的状态没有改变，那么就可以通过缓存获得当前的状态，以避免查询网络元素的系统状态的额外时间，而且可以确保状态是真实的，没有过期。

此外，OMC的启动时间也应该相对地短，通常不应该超过几分钟。为了减少启动时间，关于当前操作员对网络的视图的信息（比如经常查看的网络元素的信息）会被预先加载，以保证很快的初始访问。

Eager Acquisition模式会尽快地（通常是在启动时）从拓扑树加载同性能息息相关的资源，比如顶层网络元素的信息。这确保了资源（比如同网络元素相关的信息）可以没有延迟地被访问。

下面的顺序图描述了缓存一开始是如何以预先获得的网络元素状态信息进行填充的。在第2步，网络元素可以通过缓存快速地访问，从而为使用者避免了延迟（见图7-13）。

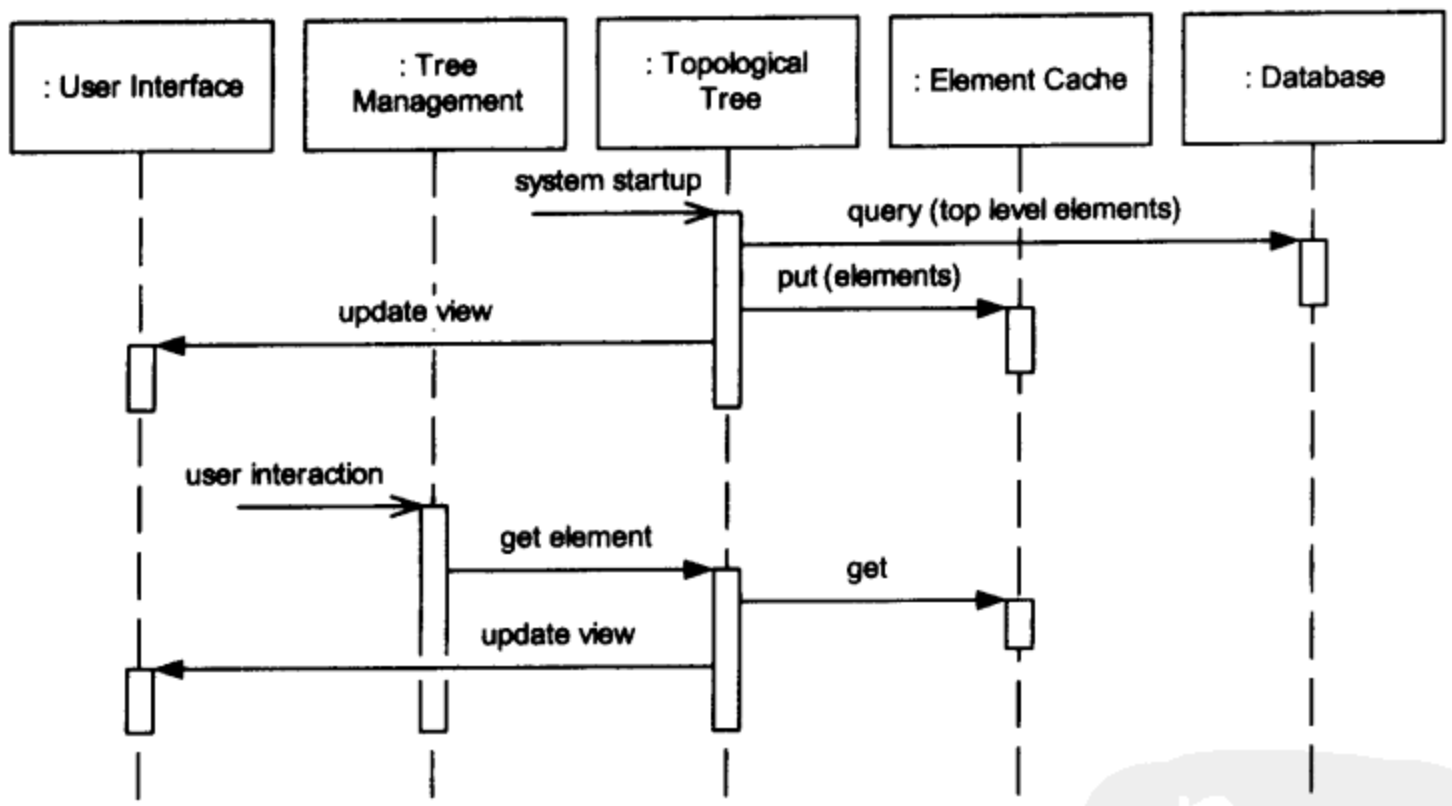


图 7-13

我们谨慎地决定了哪些信息会在启动时加载，因为Eager Acquisition模式会延长启动的时间，并且如果信息从未被访问或访问得很晚，它会造成资源的浪费。为了找出最优的平衡，执行一段时间的系统分析是很重要的。这样做的一种方式是不预先获取任何资源地运行系统，并测量多种访问时间，然后只在确实必要的情况下优化。

另一方面，有些网络元素很少会被查看。这样就可以省下内存资源的分配了（只要相应的元素不重要）。为了避免不必要地使用资源，Lazy Acquisition模式建议，只被偶然使用的资源应该在需要时（实际访问前的最后一刻）才获取。在这个例子中，存储受管对象所需的内存是资源，在内存中反映了网络元素的属性。当前的状态是延迟地从网络元素获得的。下面的顺序图中的交互描述了这一场景（见图7-14）。

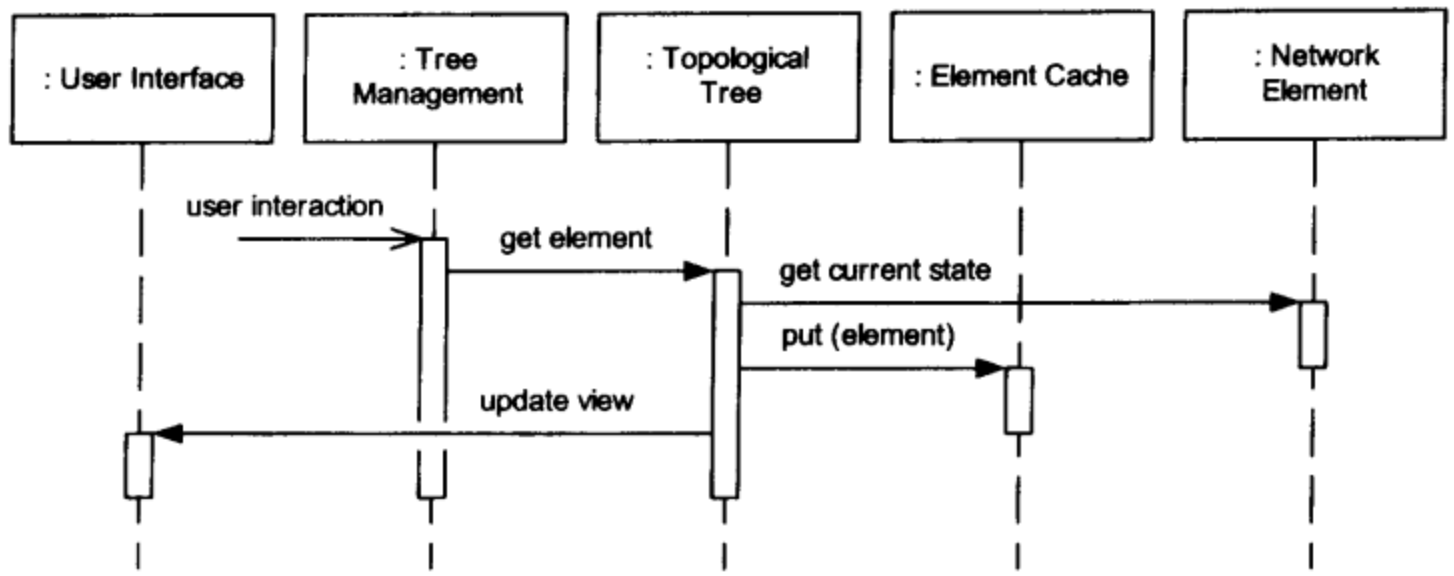


图 7-14

跟Eager Acquisiton的情况一样，应该通过系统分析来找出在哪里可以有效地使用Lazy Acquisition。

高伸缩性

OMC需要具有高度可伸缩性以应付大量的网络元素。为了让系统可伸缩，在不需要时释放资源是很重要的，这样才能让其他使用者和系统组件获取资源。当操作员对拓扑树的视图改变了，诸如此类，那么就应该释放资源。

同网络元素相关联的受管对象可能开销较大。虽然Caching模式提高了性能并且有助于在连接断开时也保持网络元素状态可获得，但是在内存中保留太多信息也会代价昂贵。因此，并不频繁需要的受管对象的状态会从内存中清除出去，以释放宝贵的资源。Evictor模式描述了如何使用这样的机制。

类似地，可以用Partial Acquisition模式来部分地获取必要的资源。当必须从网络元素获得很大的警告日志时，OMC可以先部分地从警告消息列表中获得一些最重要的记录，而延迟获取剩下的警告日志（包括不太重要的记录），直到需要时才获取。

当可视化网络元素时，图形用户界面（GUI）必须镜像出物理网络的布局。为此，对于每个网络元素都要分配一个GUI对象。当网络的视图改变了，很可能需要分配新的GUI对象。因为为这些GUI对象分配内存是昂贵的操作，需要消费内存并且带来延迟，所以应该避免这样的分配。

为了避免反复分配内存，OMC用户界面对GUI对象使用了Pooling模式来创建对象池。当分配初始数目的GUI对象时，对象池被创建。稍后随着需求的增加，对象池的尺寸会调整。当GUI对象超出了目前视图的范围时，它们就被释放回对象池。

维护和升级

OMC监控的网络元素通常会部署在很远的地方，散布在很广泛的区域，比如在建筑物屋顶上，在野外，或者在教堂的尖顶上（见图7-15）。当新的软件更新需要应用到网络元素时，让技术人员驾车到野外手工更新软件的代价太昂贵了。我们的做法是通过网络元素的O&M接口执行更新。软件改变可能也意味着协议或者策略的改变。如何确保几十甚至成百的网络元素的更新

是同步的呢?

这个问题由Coordinator模式来解决。网络元素的单个O&M组件作为软件分发和更新的整个任务的参与者。协调者的角色由OMC来扮演。这就确保了更新所有相关网络元素的任务被同步。下载到网络元素的新的软件版本只有在所有参与的网络元素都准备好激活新版本时才会激活。

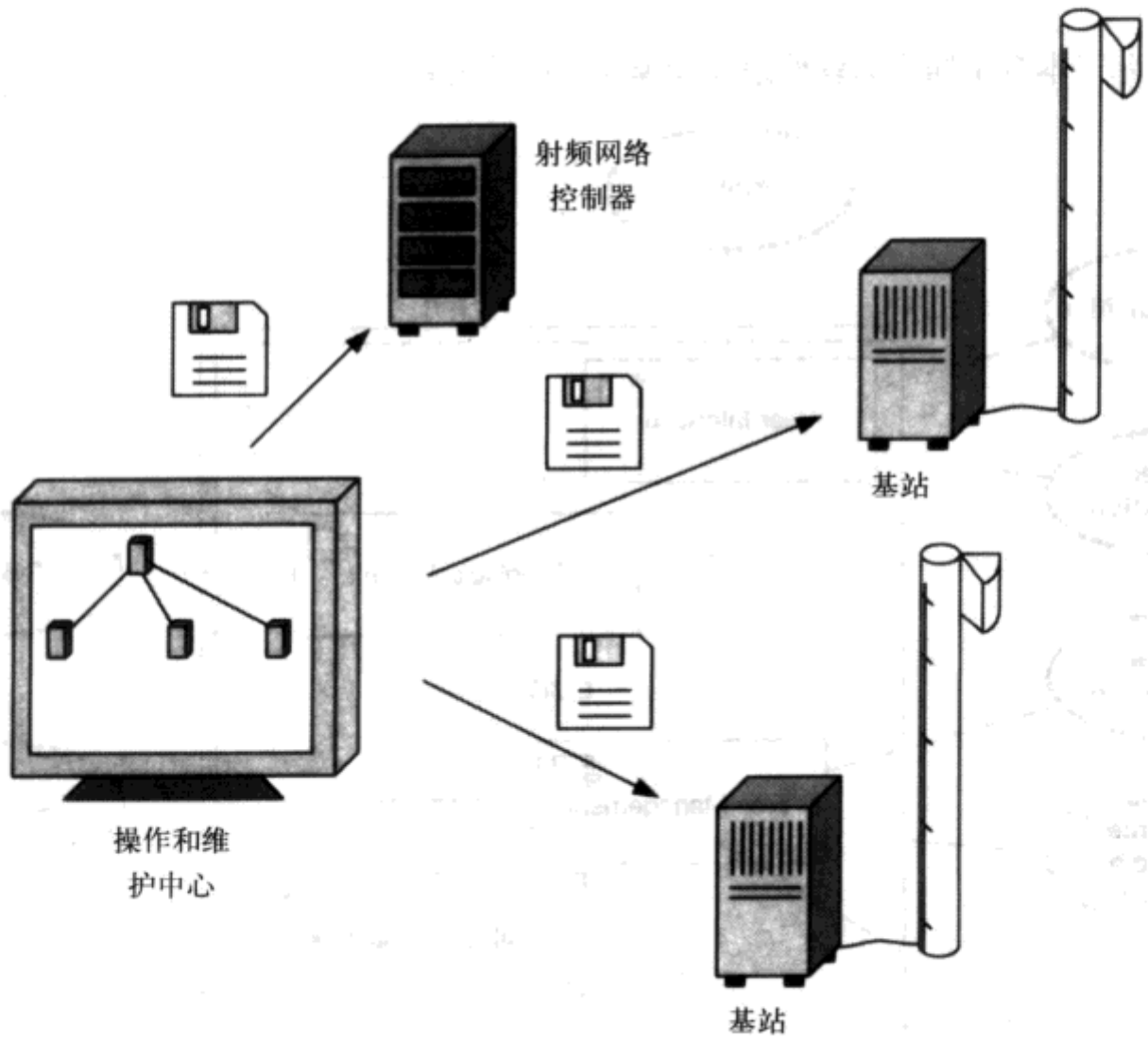


图 7-15

复杂性

前面的小节展示了OMC的操作和管理所涉及的各种资源。这些资源经常带来很强的相互依赖关系，因为必须组合使用这些资源来完成特定的任务。

思考某个持久运行的动作的例子，比如查询拓扑树以找出所有具有某个特定属性的网络元素。在这样的情形中，一个线程对多个受管对象操作，并使用数据库连接来获取它的本地数据。如果线程必须终止，那么它使用的所有资源都必须释放，受管对象和数据库连接所占用的动态分配的内存也要释放。

当需要协调多个资源时，Resource Lifecycle Manager模式的价值就体现出来了。RLM管理资源组，以必要的顺序初始化一组相关资源，并以相反的顺序释放它们。当资源变得不可用时，它还会记住如何处理故障情形。

在上面的例子中，必须首先创建一个线程，然后是数据库连接，最后是为相关的受管对象所分配的内存。如果内存的分配失败了，那么RLM先释放数据库连接并将其自动返回到池中，

然后销毁或者循环利用线程。

连接管理是使用RLM的进一步的例子，在这个例子中连接可以相互紧密关联，如果单独存在的话就毫无用处。这样RLM可以在资源组中保存关联，从而资源组知道如何处理这样的情形。

小结

图7-16展示了本章中描述的软件组件和模式之间的关系。

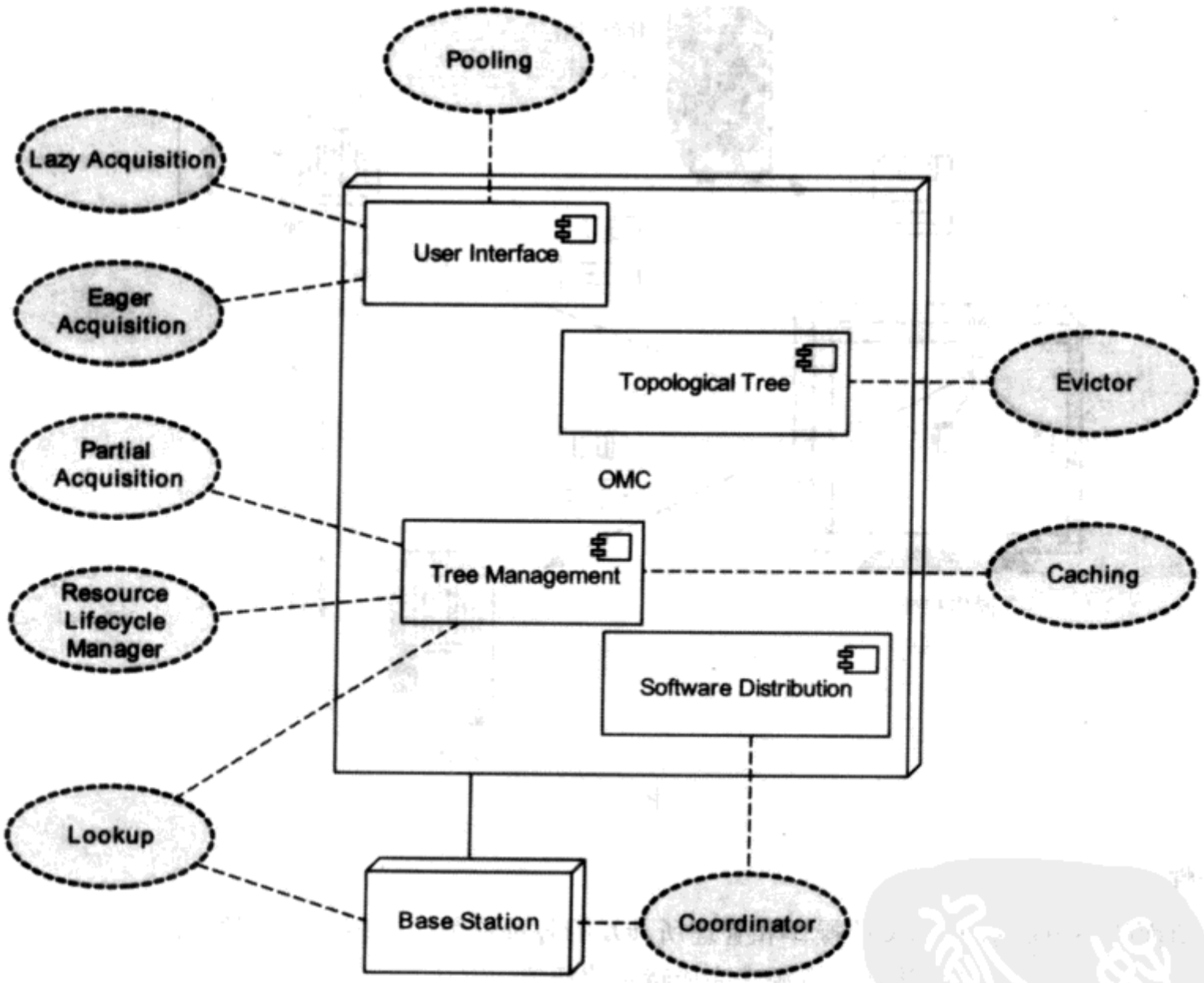


图 7-16

第 8 章
模式的过去、现在和未来

“历史很少会以正确的顺序或者在正确的时候发生，但历史学家的工作就是让它看起来是正确的。”

——James Burke

概览

第二次世界大战后德国的第一任总理Konrad Adenauer曾经说过：“Was gebe ich um mein Geschwätz von gestern”（“我不介意自己昨天的胡说八道”）。很少会有人反省自己曾经作出的对未来的预测，更鲜有人会反省两次！显然，这是合作撰写系列书的好处。

在本章中，我们将重新审视对POSA系列中第2卷《Patterns for Concurrent and Networked Objects》[POSA2]在2000年作出的对模式未来的预测。我们将讨论模式在过去4年里的方向，分析现在它们的情况，并且，借鉴后见之明，重新修正我们对模式未来的预测。

8.1 过去的四年

在《Patterns for Concurrent and Networked Objects》[POSA2]中，我们预测了我们所认为的模式发展方向。四年之后，我们可以总结一下，实际发生了什么。我们会引用很多这段时间里的重要成果（虽然我们的列表无法穷尽）。更多的引用可参见http://hillside.net/patterns/。

模式

在2000年，我们预言人们将主要在如下领域发表模式：分布式对象、实时和嵌入式系统、移动系统、业务交易和电子商务系统、现成的分布式系统的QoS、反射式中间件、优化原则、过程和组织、教育。这9个领域中，5个被我们猜对了：

- 分布式对象。今天的大多数软件系统是分布式的，特别是企业级的软件系统。因此，毫无悬念，人们在这个领域发表了大量的模式和模式语言。比如，最近的EuroPLoP和VikingPLoP会议上，有整个地关于分布式对象、分布式系统，以及其他分布方面[EPLoP01][EPLoP02][VPLoP02]的研讨会和讲座。更显著的是，出版了好几本关于这个主题的佳作。《Server Component Patterns》[VSW02]记录了描述如何用标准组件平台（比如CCM、COM+和EJB）来创建基于组成部分的分布式系统的模式语言。《Remoting Patterns》[VKZ04]描述了分布式对象计算中间件的基本组件的模式语言。《Enterprise Solution Patterns Using Microsoft .NET》[TMQH+03]记录了如何最佳地利用.NET技术；《Core J2EE Patterns: Best Practices and Design Strategies》[ACM01]则记录了如何用J2EE平台来设计系统。

- 实时和嵌入式系统。虽然今天编写的大多数软件都是为嵌入式应用而写，但是很奇怪，这个领域发表的模式不多，或者说至少没有我们所预期的多。PLoP和OOPSLA会议中有一些关于分布式系统、实时系统和嵌入式系统的研讨会，但是研讨会上介绍的模式很少有进入公众所知的模式词汇表的。这个领域最显著的工作是《Time Triggered Embedded Systems》[Pont01]的模式语言。
- 业务交易和电子商务系统。软件开发者对这个领域总会感兴趣，随着因特网和Web的流行更是如此。在这个领域，最成功的出版物可能是《Enterprise Application Patterns》[Fowl02]。它描述了如何创建3层商业信息系统，并且特别关注数据库访问层的设计。一本只专注于持久化方面的书是《Database Access Patterns》[Nock03]。还有一些出版物则专注于因特网和Web应用程序，比如《High-Capacity Internet-Based Systems》[DyLo04]的模式语言以及针对Web可用性的WU模式语言[Grah03]。
- 过程。过程模式在软件模式运动的一开始就很流行[PLoPD1][PLoPD2]。在过去的4年中，人们对此兴趣依旧，甚至越来越感兴趣[EPLoP01][EPLoP02][VPLoP02]。除了这些会议论文，还有3本有价值的关于过程及过程相关模式的书出版：《Object-Oriented Reengineering Patterns》[DDN03]、《Domain-Driven Design》[Evans04]以及《Configuration Management Patterns》[BeAp02]。所有这些书都会告诉你相应领域的“what”、“when”和“why”，并且同一些当代的软件开发过程文献相比，提供了一些真正的价值。
- 教育。有一个小而活跃的社区在它自己的网站[Peda04]发表教学模式。这个社区还在模式会议上主办关于教育和教学的讲座[EPLoP01][EPLoP02][EPLoP03]。这个社区的成员撰写的所有模式要么是关于软件开发的人的因素，要么是关于人如何学习或者如何教学。

在其他四个领域（移动系统、现成的分布式系统的QoS、反射式中间件、优化原则）几乎没有模式被发表。虽然对于后三个领域或许没什么值得惊奇，因为它们要么很窄，要么大多数开发者对其兴趣有限。但是，对于人们缺少兴趣发表关于移动系统的模式则没有合理的解释。移动性对于很多系统以及可能需要新项目的场景（比如“商务旅行者”、“移动办公室”、“移动工作者”、“空中企业”等）都是个重要的话题。可能在不久的将来，人们会对移动性模式产生更多的兴趣，从而使得我们2000年的预言最终成真。

除了我们的预测领域，还有很多其他领域也发表了模式，其中有一些我们本来或许可以预测到的：

- 资源管理。资源管理是一个重要的主题，不仅对于嵌入式系统如此。你现在在读的这本书覆盖了这个领域的重要方面，从资源获取到资源使用再到资源释放。
- 故障处理。2000年之前就有人记录故障处理的模式了[ACGH+96]，但是这个领域近两年来又获得了模式作者的关注。这个领域的很多最新进展都以模式的形式表示出来了[Sari02][Sari03]。
- 安全。安全在模式社区变成了一个极热的话题。几次EuroPLoP会议都开展了针对安全模式的研讨会，笔者的研讨会上也介绍了很多安全模式[EPLoP02][EPLoP03]。
- 消息传递。最近这个领域出版了一部重要著作：《Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions》[HoWo03]。这本书以模式的形式

记录了所有你需要知道的关于消息传递的东西。它可能是过去四年中出版的最成功的一本模式著作。

- 人机交互。对人机交互模式的兴趣以前就存在[PLoPD2][PLoPD3]。但是，在过去的4年中，很多交互模式都在近期的模式会议上发表了[EPLoP02][EPLoP03]，并且有了一个关于人机交互模式的网站[HCI99]。这个领域的第一本模式书是《A Pattern Approach To Interaction Design》[Borc01]，随后就是关于Web可用性的WU模式语言[Grah03]。

虽然我们的一些预测没有成真，但是正如这个汇总所证明的，我们对模式的整体预测还是准确的。在过去的4年里，只有少量的通用模式著作发布，大多数出版物都是关于特定领域的模式的。只有一个例外：我们完全低估了Gang-of-Four [GoF95]的威力。确切地说，我们没料到会有那么多GoF模式的相关著作出版。这些著作要么把GoF模式翻译到其他编程语言[Metz02]，要么更深入地解释这些模式的一些特定方面和值得告诫的地方[ShaTr01][Henn04]。

模式语言

在我们2000年的预言中，最重要的声明是模式语言会越来越多并且越来越重要。我们预测了3个具体的趋势，并且很高兴地看到它们都成为了现实：

- 新的模式著作将主要以模式语言的形式出版。事实上，我们在前面章节引用的大多数文献都是由模式语言组成的。
- 现有的独立的模式，比如来自Gang-of-Four [GoF95]和《A System of Patterns》[POSA1]的模式会整合进现有的以及新的模式语言。分布式计算的模式语言在最近的EuroPLoP会议上发布了一部分，并且会被包含进POSA系列的第4卷[POSA4]，它可能是这些语言的最显著的代表。它包含了超过100个模式，大多数都是已知的独立模式。另一个整合了很多著名的独立模式的模式语言是《Enterprise Application Patterns》的语言[Fowl02]。
- 庞大、复杂，或者超过正常尺寸的独立模式会被分解成更小的、更实际的模式语言。这个预期也成为了现实。例如，《Remoting Patterns》[VKZ04]就把《A System of Patterns》[POSA1]中的Broker模式分解成了细节。POSA1中的另一个模式Pipes and Filters则在《Enterprise Integration Patterns》[HoWo03]被分解了。最近，一些Gang-of-Four模式[GoF95]也被分解成了更小的模式群体[HeBu04]。

经验报告、方法和工具

我们在这个领域的预言在很大程度上失败了。一方面，有一些使用模式的经验报告被发表[Busc03]，并且也发展出了一种应用模式的新方法[YaAm03]；另一方面，无论是经验报告还是《Pattern-Oriented Analysis and Design》方法都没有吸引模式社区的很多注意力。所以，它们的影响非常有限，这和我们对这个领域的预期不符。

模式记录

很少有关于如何记录模式的内容发表。所以，我们的预测再次失败了。

模式和模式语言的形式化

这是我们的预测的又一个失败，虽然我们没有想到会这样。在20世纪90年代末期，模式的形式化很流行。但是，在那之后，模式形式主义者阵营就平静下来了。这可能是人们逐渐认识到，要形式化模式几乎是不可能的，而且过去的形式化表达模式的方法即便要么不完整要么很有限，也还是太复杂而没有实用价值。

8.2 模式的现状如何

这是个很有趣的问题。事实上，这个问题比4年前更难回答，比1996年则难答得多。当时我们做出了关于模式的未来的第一批预测[POSA1]。

毫无疑问，今天模式已成为主流。几乎所有的软件系统都包含模式，它们经常被单独使用以解决特定的问题，也有时候整体地应用，基于模式来设计和实现整个系统。模式社区原来的意图“传播关于良好的软件工程实践的知识”几乎已经实现了。

至于领域的覆盖，我们在过去的4年中看到了明显的进步。十年前，几乎所有发表的模式都是通用的，分布式或者事务处理之类的特定领域根本没有被涉及。4年前，发表了第一个领域相关的模式，但是实际涉及的领域还是只有少数。今天，情况已经大不相同了。模式和模式语言已经在多个领域发表，包括（但不限于）：分布式、同步、资源管理、事务处理、消息传递、安全、故障管理、事件处理、用户界面设计、电子商务。我们很高兴看到对于用模式记录经验和最佳实践的越来越多的兴趣，因为这证明了整个想法的质量和表达力。

关于模式的理解，我们看到了两个“阵营”。幸运的是，同4年前相比，绝大多数软件工程师对模式的理解都要有了很大提高。但遗憾的是，软件社区的太多成员还在蹒跚于常见的陷阱，或者成了对模式的常见误解的牺牲品。关于模式的极其成功的出版物《Design Patterns Java Workbook》[Metz02]和《Beyond the Gang-of-Four教程》[HeBu04]体现了这一事实，所以，对于模式究竟是什么我们需要进一步的更精深的教育。

最后（但并非最不重要），发表的模式以及模式语言的质量有了显著提高。现在模式社区可以依赖十年来使用和撰写模式的经验，模式作者对于用模式的形式来记录他们的知识变得更有经验。我们在8.1节引用到的大多数模式和模式语言都比过去发表的模式和模式语言更具表达力，更精确，可读性更好。

8.3 模式明天将走向何方

在软件社区，模式依然同开始时一样流行：模式会议依然众多，在其他会议上发布的关于模式的演讲和辅导也很受欢迎，并且越来越多的开发者报告说他们在日常工作中使用模式。换句话说，模式已经成了我们软件开发技术“百宝箱”中的“日用品”了。因此，很可能在将来模式会被进一步研究，会进一步演化，以便深化对概念的认识和理解，支持软件工程师对模式的使用，以及对软件开发中尚未探索的领域加以模式化。

模式和模式语言

我们对模式的主要预期是，过去4年的两个主要趋势将会延续：作者们将主要关注于记录模式语言而不是单独的模式，大多数新的模式和模式语言都将是领域相关的，而不是通用的。我们预期会记录模式和模式语言的一些领域是：

- 安全。从这个千年的开始，安全就是模式的一个重要主题，但是最近这个领域的关键人物在EuroPLoP会议上会面了，共同为这个主题的更完整覆盖而工作。因此，我们不仅预期在安全相关领域（比如验证、授权、完整性和机密性）会有新的模式和模式语言被记录，我们还预期现有的模式和模式语言会被整合以提供安全应用程序的更一致的模式视图。
- 容错和故障管理。在模式社区的初创时期，容错和故障管理是模式作者间的流行话题[ACGH+96]，但是5年多来，这个领域几乎什么都没发表。直到最近，人们对这个主题的模式才增加了兴趣[Sari02][Sari03]，可能是对越来越多的对通常的容错和故障管理的兴趣做出的回应。人们还没意识到并广泛承认，软件质量（特别对于大规模和分布式系统）需要合适的故障管理概念才能真正成功。基于这个理由，我们认为，正如在安全领域那样，专家们将对此领域做出更有深度的模式覆盖。
- 分布式系统。分布式系统是（并且将会继续是）模式作者的主要领域。虽然模式已经覆盖了很多分布式系统的主题，但是还有一些空白。例如，对等计算和网格计算是两个很热的概念，但是相关的模式很少或者根本没有。不过，我们看到关于这些领域以及其他同分布式系统相关的主题的模式和模式语言发表只是时间问题。
- 移动性。虽然我们关于“移动性将是个诞生新模式和模式语言的领域”的预言曾经失败过一次，但是我们还是要再这样预测一次。理由很简单：对移动系统的需求增长得很快，而且移动系统会面临很多在非移动系统中不存在的问题，比如管理不足以及可变的带宽及电能，或者适应连接性和服务及性能的频繁破坏。它一定会成为模式作者非常感兴趣的领域。因此，我们预期，有经验的移动系统开发者会把他们的专业知识记录成模式的形式以表示这个领域的最佳软件开发实践。
- 普及计算。“普及”、“环境”、“无处不在的”以及“感知”都是你在关于现代软件技术的会议和其他论坛上可以经常听到的术语。但是，建立这样的系统非常具有挑战性。不仅一些重要的领域（比如嵌入式系统、实时系统、移动系统）与此相关，而且普及系统还有其自己的挑战。例如，普及系统不能通过专门的管理点来集中维护和配置，而必须自行安装、自行管理，以及在可能的情况下自行修复。再比如，因为普及系统的意图是支持很多日常生活中的活动，包括家庭自动化到个人健康控制，所以它们会直接地、广泛地影响我们的个人习惯。因此，普及系统必须设计成不管何时何地只要需要就能提供服务，而另一方面又会尊重你的隐私而不会试图控制你的生活方式。掌握这个“助手”和“老大哥”之间的微妙的平衡既需要技术经验，也需要社会经验。随着这样的系统越来越多，我们预期开发者会以模式的形式记录这样的问题的解决方案。
- 同语言相关的惯用法。在过去几年里，新的编程语言和风格被开发出来，并获得了一定的流行度。通用语言的例子包括Python [MaAs02]和Ruby [ToHu00]，AspectJ [Ladd03]则是特殊目的的语言，还有产生式编程[CzEi00]的范型。这些新语言和风格都有其自己的惯用法，

这将各种语言的编程区分了开来。对于C++、Java、C#和Smalltalk，人们已经记录了这样的惯用法，但对比较非主流的语言和风格（比如前面列出的那些），还没有这样的记录。我们希望这个空白可以被填上，以便为这些语言提倡好的编程实践。这是否会发生取决于这些语言和风格从软件社区获得了多少关注。如果它们始终是少数圈内人士的关心内容，那么可能就不会发生，但如果更多的开发者关注它们，那么我们就可以预期会有人收集并发表这些惯用法。

- 过程。虽然软件开发的很多领域都被模式所描述了，但是也有些角落尚待发现。例如，在敏捷过程的领域，测试驱动开发或者单元测试这样的方法浮现出来，并且人们获得了对支持敏捷性的项目和组织结构的新认知。模式社区过去偶尔也涉及了一些这样的主题[Cope95]，但还没有透彻的模式覆盖。所以，我们预期在将来的几年中会有更多关于过程和组织结构的模式和模式语言。
- 教育。所有主要的模式会议都会举行关于教育的模式的讲座。看起来就像是模式和模式语言成为了教师和咨询者交换关于编程和软件工程教学知识的标准载体。虽然过去发表了很多教学模式，但其他会议上举办的关于教育和教学的研讨会表明，人们还需要更多的模式。我们预期这个需要会被满足。

很可能模式和模式语言还会在除了这些领域之外的其他领域发布，但是基于我们目前的知识和所看到的现状，这些是最有可能的领域。希望我们能有机会在将来重新审视我们的预告，以检查哪些预言成真了、哪些预言失败了。

理论与概念

今天模式社区在发掘、记录和使用模式方面已经有了超过十年的经验。很多有意识地使用模式的软件系统都获得了成功，但也有很多失败的故事。现在是重新审视过去并发展出对模式的一般理解的时候了：它们（究竟）是什么，它们有什么样的属性，它们的目的是什么，应该谁来使用它们，使用它们时什么应该做，什么不应该做。类似地，既然现在模式社区广泛地使用了模式语言，那么就需要合适的概念基础来避免对这种模式组织形式的误解和误用。

这样的模式理论并没有形式化，还没发现表现模式想法的形式方法，目前只能用描述的方法。我们的目标不是做到像数学般地绝对精确，而是要支持对软件开发及其最佳时间的理解。

这一理解在当今世界的半形式化和工具支持的软件工程中基本上不存在。结果就是，虽然有形式化和工具的“支持”，但是很多软件开发项目依然失败了。模式社区的一位著名成员Kevlin Henney曾经说过：“打字从来不是软件开发的瓶颈，理解才是”。

因此，我们预期未来几年会有关于模式和模式语言的概念的书和论文发表，但是，遗憾的是，这种著作的可能的作者范围相当小。这毫无疑问需要撰写和应用模式及模式语言的专家经验，但模式社区只有极少数成员可以声称“达到了这一程度”[Alex79]并对此概念具有相当熟悉的程度而可以写出我们正在寻求的“编程的永恒之道”。

重构和整合

在过去的十年中，发表了很多的模式，其中很多模式成了几乎所有软件开发者的设计词汇

表的一部分。但是，这些模式的原始描述并没有覆盖自从它们发表之后软件开发的进步。因此，有一些模式需要修订或者重构以发展到设计和编程的现有知识水平。模式社区注意到了这个问题，并且开始重构一些著名的模式以表述我们今天对软件开发的认识[Metz02][HeBu04]。我们预期在将来可以看到更多的模式重构，特别是经典模式著作中发表的模式。

模式社区的另一个重大趋势是整合。如我在本章一开始提到的，未来大多数模式出版物都将是模式语言。这些语言不仅包含全新的模式，还会基于现有的模式工作而创建。例如，几乎近年发表的所有模式语言都在某种程度上联系到了GoF的经典模式。这不仅增加了较早的模式的单独价值，还把新模式和较早的模式联系成了一个模式网络，这个网络跨越了很多的模式语言，跨越了软件开发的很多领域。换句话说，我们更加接近了模式社区原来的愿景之一：提供一种“宏大的软件构架手册”。虽然我们离这个愿景还有千里之遥，但每种整合了现存模式成果或者基于现有模式成果而创建的模式语言都是“征程”中前进的步伐。

Gang-of-Four

虽然已经过去了十年，但是Gang-of-Four的书[GoF95]依然是最有影响的模式著作。毫无疑问，这本书是开创性的，并且影响了我们很多人对软件设计的思考。但是，自从1994年之后，一切并没有停止。今天，我们对模式和一般的软件开发知道得更多，也对Gang-of-Four模式有了更多的认识。于是，无论是从内容上还是从概念问题上，我们都理应对这些模式进行一些更深入的解释，并做一些重构。

过去已经出版了一些关于Gang-of-Four模式的书和论文，但我们还看不到这一趋势的尽头。我们预期，关于Gang-of-Four著作的沉思将会带来一系列的衍生著作，比如如何用其他编程语言来实现这些模式，关于缺少的元素以及替代解决方案的文章，模式的正确范围，以及对某个模式是否名副其实、某个模式是不是该退休了的讨论。Gang-of-Four是个永不结束的故事。

8.4 关于模式未来的简短声明

虽然我们预期本章中的很多预言会成为现实（因为它们来自对过去和现在的深入分析），但是这些预期也包含了不确定性。

模式的未来可能会沿着我们今天没想到的方向前进。因此，请把本章中的预测当做持续进行的关于“模式和它们的未来”的对话中出现的很多可能愿景中的一个，而不要认为这是永远正确的先知的预言。

PDG

第9章 结　语

“人类智慧的总和并不包含在任何语言中，也没有哪种语言可以表示所有形式的人类知识。”

——Ezra Pound

本书中展示的大多数模式都在一定程度上概括了已经存在的模式。例如，Pooling模式可能有很多特定的实例，比如Thread Pooling和Connection Pooling。我们写本书的目的之一是找出这些模式之间的共同点和不同点。因此，这些模式不仅仅是基于我们的经验，还基于很多研究者和开发者的经验。

如我们所试图展示的那样，本书中的模式并不特定于任何一个领域。有效且高效地管理资源是一个跨越领域和系统，也跨越了系统中的层次的挑战。在找出资源管理模式的时候，我们试着利用了自己创建很多不同领域不同尺寸的系统的经验。我们的经验包括创建实时嵌入式系统、自治式系统、基于Web和大规模的企业系统。虽然我们创建的系统的领域和类型变化很大，但是资源管理的基本问题很类似。这激励着我们把解决方案以模式的形式写下来，我们觉得这种形式最能说明问题。

随着越来越多的语言和运行时环境支持垃圾收集的概念，这使得运行时系统可以自动检测和释放不再被使用的内存。垃圾收集是Evictor模式的一个特化实例，它包含了支持自动识别不使用的资源的逻辑。通过识别不使用的资源并释放它们，垃圾收集为运行时环境带来了更多的控制和关于资源可用性的更多可预测性。你除了可以发现Java和C#在运行时环境中实现了垃圾收集外，还可以找到垃圾收集的C++实现[Boeh04]。可以观察到，随着对显式资源释放放宽要求的中间代码越来越常见，今天的基础环境越来越要求支持自动且有效的资源释放机制。

J2EE[Sun04b]和.NET[Rich02]之类的中间件技术的JVM（Java虚拟机）和CLR（公共语言运行时）分别把很多本来开发者需要面对的冗繁的资源管理工作自动化了。这些技术在基础设施中实现了很多本书中描述的模式。Pooling模式和Caching模式被用来使得EJB容器以及JSP和ASP.NET引擎变得高效。.NET Data Set使用了Caching模式来使我们可以高效地使用数据库内容。使用这个模式，.NET Data Set避免了昂贵的对数据库的访问，因为它为数据库客户提供了相关数据的本地拷贝。.NET还使用了Leasing模式。比如，在.NET Remoting [Ramm02]中，客户激活的远程对象是基于租约来管理的。

正在浮现的最重要的下一代技术之一就是“网格”计算[Grid03]。网格计算是关于分布式资源（比如处理时间、存储空间和信息）的分享和聚合的。一个网格包含多台相连接以形成单一系统的计算机。Lookup模式和Resource Lifecycle Manager模式很适合解决网格计算的一些需求。两个模式都在连接和管理多个资源方面扮演了重要的角色。网格可以看做是一个大型的资源管理系统，在这个系统中有效地管理处理器时间、内存、网络带宽、存储空间以及其他类型的资源很重要。总体的目标就是有效且高效地调度需要使用网格环境中可用资源的应用程序。有一

些网格（比如Condor[Cond04]）使用了Coordinator模式来同步任务的提交和完成。可用的网格计算项目[BBL02]和网格工具的构架各不相同，但是其中大多数的本质概念还是一样的。在很大的程度上，网格计算环境的构架同自组式网络和对等（P2P）网络的构架很类似。更多细节请参见第6章。

除了网格计算外，自主计算的领域近来也很受关注，在这个领域有很多研究在开展。根据人体的自主神经系统建模，自主系统的目标是控制软件的行为而不需要用户输入。自主系统会管理自己，并且必须能够以自动的方式来管理它的资源。这可能会带来一组新的需求和挑战，比如自动资源预留、资源分配和重新分配，以及资源的移动。自主系统需要预期所需的资源，同时还要隐藏其复杂性。此外，自主系统通常被看做能够自愈，具有自动从失败中恢复的能力。这样的行为和功能也需要有效的资源管理。

为了解决网格计算和自主系统带来的挑战，我们的资源管理模式语言很可能需要扩展。虽然本书中展示的模式可以处理这些形成中的技术的很多需求和作用力，但是可能还会发现其他的模式。要记住这一点，我们目前的贡献只应被看做是寻找更大的资源管理模式语言的开端。

我们希望本书是资源管理模式语言的开始，它将会随着资源管理的知识普及以及隐藏因素变得可见而成长。我们有意识地没有涉及一些同资源管理相关的领域，比如容错系统[Sari02][ACGH+96]中的资源的管理，负载平衡的影响，以及整个引用计数[Henn01]领域。

对于上面提到的技术，以及随着将来对资源管理的关注的增多，同资源管理相关的领域将如何形成和发展模式语言将是令我们非常兴奋的话题。

引用到的模式

下面对本书中提到的模式给出了简略的描述。

Abstract Factory（抽象工厂）模式[GoF95]提供了一个接口，可以创建相关或者互相依赖的对象家族，而不需要说明它们的具体类。

Abstract Manager（抽象管理器）模式[Lieb01]专注于管理企业系统中的业务对象。

Activator（激活者）模式[Stal00]有助于高效地实现按需激活和停止被多个客户访问的服务。

Active Object（主动对象）模式[POSA2]解除了函数的调用和执行的耦合。

Adapter（适配器）模式[GoF95]把一个类的接口转换成客户期待的接口。Adapter让本来因为接口不兼容而无法共同工作的类变得可以共同工作。

Asynchronous Completion Token（异步完成令牌）模式[POSA2]允许应用程序高效地分发和处理来自它调用的服务的异步响应。

Cache Management（缓存管理）模式[Gran98]专注于如何缓存Java对象以及把缓存同Manager模式[Somm98]相结合。

Cache Proxy（缓存代理）模式[POSA1]在代理内部实现了缓存，它表示一个或多个客户想要获取数据的数据源。

Comparand（被比较字）模式[CoHa01]提供了在特定环境中对不同对象做相同解释的方法。它在每个感兴趣的类中引入一个实例变量（被比较字）并用它来比较，从而实现了这一点。当多个引用指向概念上的相同对象时，建立不同对象的“相同性”是必要的。

Component Configurator（组件配置器）模式[POSA2]允许应用程序在运行时链接组件实现或者解除链接，而不必修改、重新编译或者静态地重新链接应用程序。

Command Processor（命令处理器）模式[POSA1]把服务的请求同它的执行相分离。命令处理器组件把请求作为单独的对象来管理，并调度它们的执行，并且提供额外的服务，比如储存请求对象以便稍后撤销请求。

Data Transfer Object（数据传输对象）模式[Fowl02]在远程客户和服务器端之间传输数据（比如对象或者调用参数）。这个模式提供的封装减少了传输这类数据所需的远程操作数目。

Deployer（部署者）模式[Stal00]描述了如何配置、部署和安装软件工件。

Disposal Method（清除方法）模式[Henn03]封装了对象清除的具体细节，这是通过提供显式方法来清除对象而不是直接把对象扔给垃圾收集器或者删除它们来做到的。

Double-Checked Locking Optimization（二重检查加锁优化）模式[POSA2]降低了在程序执行时代码临界区必须获得一次锁（为了满足线程安全性）的竞争和同步开销。

Factory Method（工厂方法）模式[GoF95]定义了创建对象的接口，但是让子类去判断实例化哪个类。工厂方法让类可以把实例化推托给子类。

Fixed Allocation（固定分配）模式[NoWe00]通过在程序初始化的时候分配好必需的内存来

使得内存消耗具有可预测性。

Flyweight（享元）模式[GoF95]通过共享来高效地支持大量的细粒度对象。

Half-Object Plus Protocol（部分对象协作协议）模式[Mesz95]在对象被两个分布式客户使用的时候，把对象的职责分割成两个部分，并把它们分派给两个相互依赖的部分对象。出于效率的考虑，每个部分对象都实现了本地用得最多的职责。这一模式让部分对象通过某种协议来协作。

Interceptor（拦截器）模式[POSA2]允许服务被透明地加到某个框架并在当特定的事件发生时自动触发。

Lazy Load（延迟加载）模式[Fowl02]把数据从数据库的加载延迟到第一次访问的时候。

Lazy Optimization（延迟优化）模式[Auer96]只有正确地确定了设计之后再优化这部分软件的性能。

Lazy Propagator（延迟传递）模式[FeTi97]描述了在相互依赖的对象网络中，对象如何判断是否受其他对象的状态改变的影响，从而需要更新自己的状态。

Lazy State（延迟状态）模式[MoOh97]把对象状态[GoF95]的初始化延迟到了访问状态的时候。

Manager（管理器）模式[Somm98]把应用于一个类的所有对象的功能放到了单独的管理对象中。这样的分离使得管理功能可以单独变化，也使其可以被不同的对象类所重用。

Master-Slave（主从）模式[POSA1]支持容错处理、并行计算和计算准确性。主组件把工作分派到相同的从组件，并从从组件返回的结果计算出最终的结果。

Mediator（协调者）模式[GoF95]定义了封装一组对象交互方式的对象。协调者有助于耦合的松散化，因为它组织了对象显式地互相引用，让它们的交互可以独立地变化。

Memento（备忘录）模式[GoF95]用一个单独的、可持久化的对象封装了另一个对象的状态。

Model-View-Controller（模型-视图-控制器）模式[POSA1]把一个交互式的应用程序分成了3个组件。模型包含了核心功能和数据，视图则向用户显示信息，控制器则处理用户输入。视图和控制器共同组成了用户界面。变化传递机制确保了用户界面和模型的一致性。

Object Lifetime Manager（对象生命周期管理器）模式[LGS01]是专门用于在没有正确静态析构函数的操作系统（比如实时操作系统）中管理单件对象的。

Object Pool（对象池）模式[Gran98]管理一类创建起来很昂贵或者可创建数目很有限的资源的重用。

Page Cache（页面缓存）模式[TMQH+03]在访问动态生成的Web页面的时候改善了响应时间。页面缓存同用其来保存被访问过的页面（通过URL来索引）的Web服务器相关联。当请求相同的URL时，Web服务器查询缓存，并返回缓存的页面，而不是再次动态生成其内容。

Passivation（冻结）模式[VSW02]把组件实例的内存表示持久化存储，或者从持久化存储中读入组件实例的内存表示并激活组件。

Pooled Allocation（池式分配）模式[NoWe00]预先分配一池的内存块，并且在内存被返回后循环利用。

Proxy（代理）模式[GoF95]提供了另一个对象的代理或者占位符，用以控制对其的访问。

Proactive Resource Allocation（前摄资源分配）模式[Cros02]会预期系统的变化并预先计划必要的资源分配，目标是即便在变化的条件下也要维持系统性能。

Reactor（反应堆）模式[POSA2]使得事件驱动的应用程序可以把来自一个或者多个客户的服务请求分发出去。

Reflection（反射）模式[POSA1]提供了动态改变软件系统的结构和行为的机制。一个元层次提供了关于所选择的系统属性的信息，使得软件可以获得自身的信息。

Resource Exchanger（资源交换器）模式[SaCa96]通过在分配和使用资源时共用在客户端和服务端之间的公共缓冲区来降低服务端的负载。

Singleton（单件）模式[GoF95]确保一个类只有一个实例，并提供了全局访问点。

Sponsor-Selector（赞助商-选择器）模式[Wall97]把3个基本的职责分离开来：推荐资源、选择资源和使用资源。

State（状态）模式[GoF95]使得对象在内部状态改变的时候可以改变其行为。

Strategy（策略）模式[GoF95]把算法之类的逻辑封装成独立于客户请求的可互换的类。

Thread-local Memory Pool（线程专属内存池）模式[Somm02]允许为每个线程创建一个内存分配器。这有助于降低同步开销，因为动态内存分配是由预先分配好的内存线程局部的内存池来执行的。

Thread Pooling（线程池）模式[PeSo97]描述了如何约束使用线程的数目以及如何复用不用的线程。

Variable Allocation（可变分配）模式[NoWe00]通过执行按需内存分配来优化内存消耗。

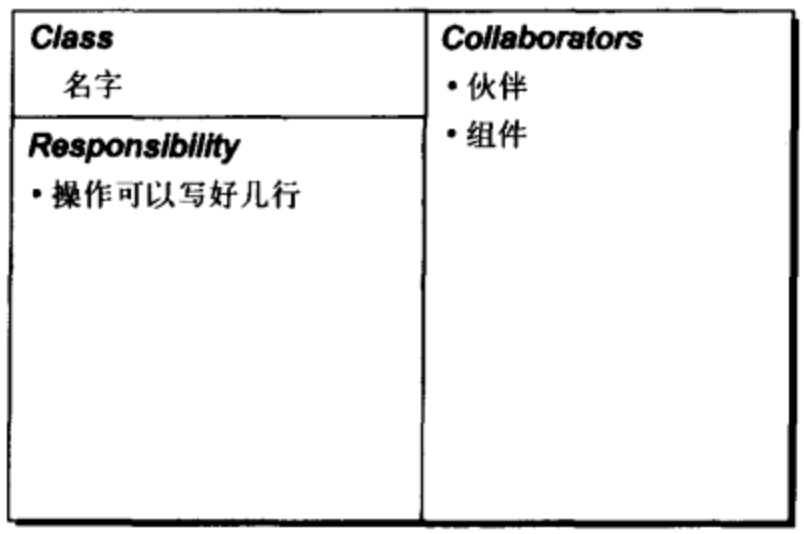
Virtual Proxy（虚拟代理）模式[GoF95]根据需要加载或者创建代理所表示的对象。

Wrapper Facade（封装门面）模式[POSA2]用更简洁、健壮、可移植、可维护、内聚的面向对象类接口封装了现有的非面向对象API所提供的函数和数据。

符号表示法

CRC卡片

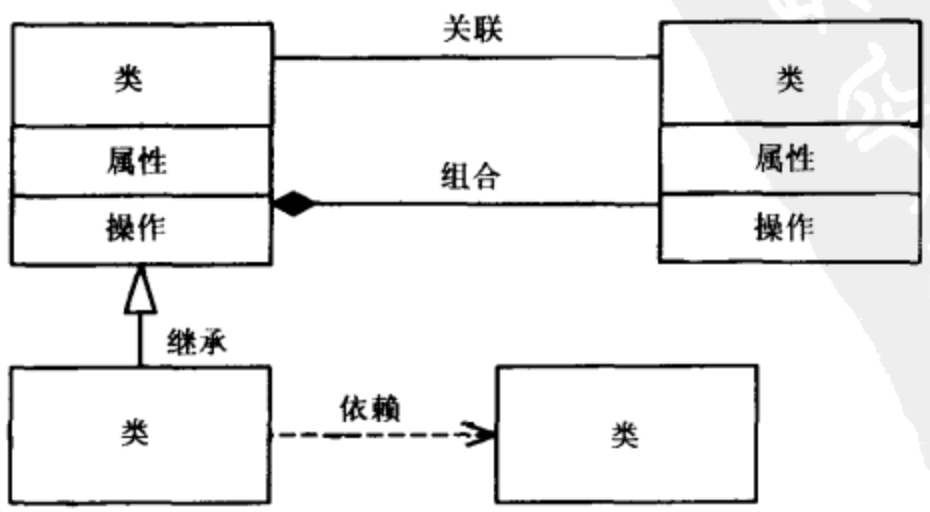
CRC（Class-Responsibility-Collaborators，类-职责-协作者）卡片[BeCu89]有助于辨识并以非正式的方式说明应用程序，特别是在软件开发的早期阶段。

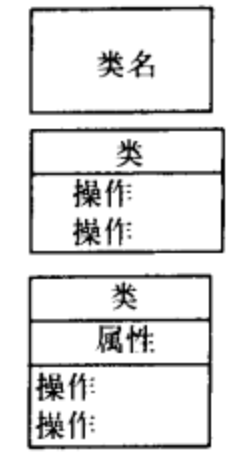


CRC卡片用来描述实体，比如组件、一个对象或者一类对象。卡片包含3个区域，分别描述了实体名称、职责和其他协作实体的名称。使用术语“类”是出于历史原因[Rees92]。

UML类图

UML（统一建模语言）[Fowl03]被广泛用于软件应用程序的分析与设计。类图是一种重要的UML图。类图描述了系统中对象的类型，以及对象之间的多





种静态关系。

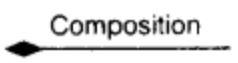
类（Class）。描述类名及其（可选项）属性和操作的矩形框。抽象类以及相应抽象方法的名称是用斜体来表示的。

操作（operation）。操作名写在类框中。它们表示类的方法。抽象操作（也就是只为多态提供接口的操作）用斜体表示。

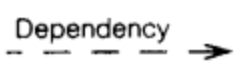
属性（attribute）。属性名写在类框中。它们表示类的实例变量。



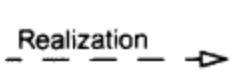
关联（Association）。连接类的线。关联可以是可选的，也可以是多重的。关联端的数字表示关联有几重。类的关联用来表示类之间任何不属于聚合、组合或者继承的关系。



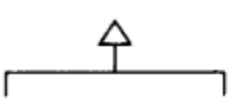
组合（Composition）。关联端的实心菱形表示另一端的类是这个类的一部分，拥有相同的生命周期。



依赖（Dependency）。依赖表示某个类为了某个目的使用了另一个类的接口。



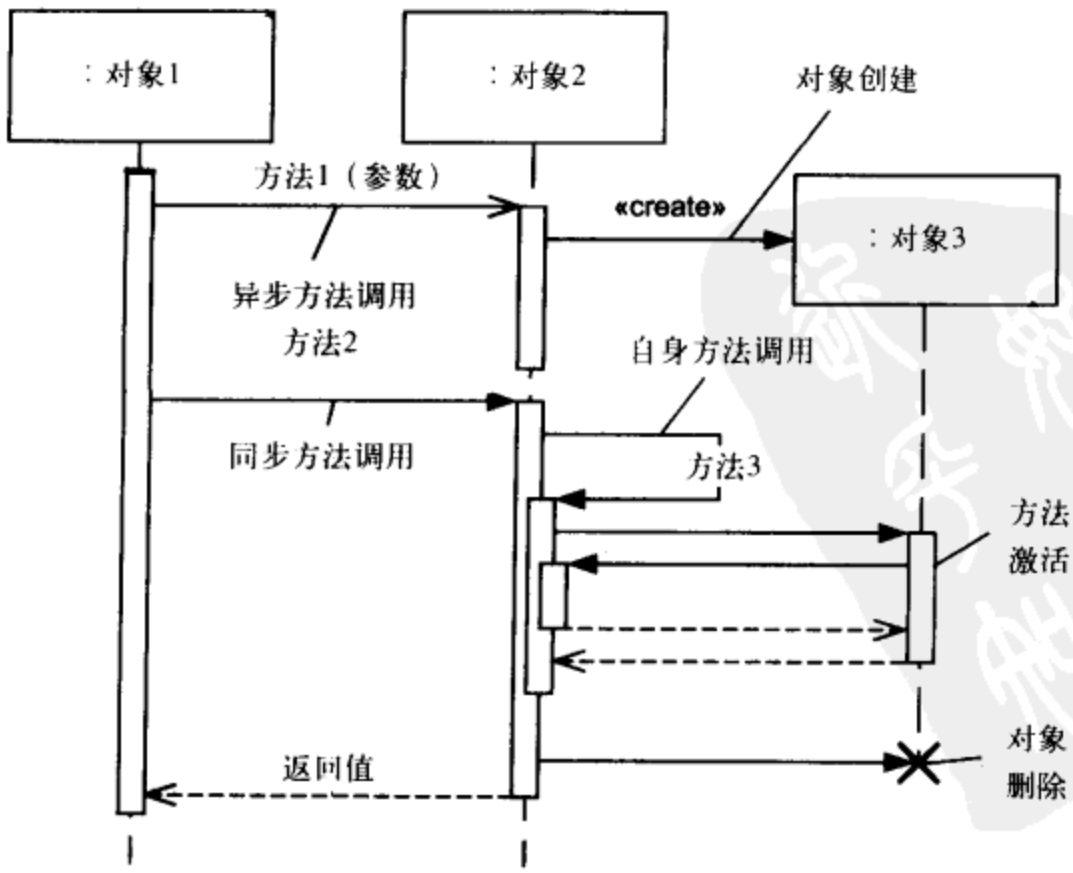
实现（Realization）。实现（realization）表示规约和实现（implementation）之间的关系。

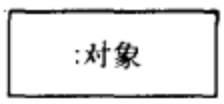


继承（Inheritance）。继承是用关联线顶端的三角形来表示的。三角形的顶点指向超类。

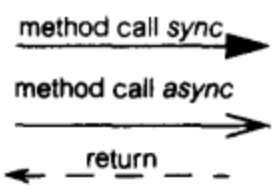
UML顺序图

UML顺序图描述了对象组如何在某个行为中协作[Fowl03]。





对象（Object）。顺序图中的对象或者组件被画为矩形框，并且用模式中的组件的名字标记。顺序图中发送或者接受方法调用的对象在方框底部会有一条竖线。

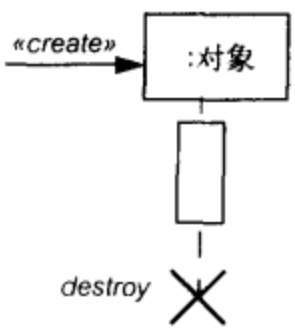


时间（Time）。时间从顶向下流动，并用虚线表示。时间轴没有刻度。

方法调用（Method Call）。对象之间的方法调用用箭头表示。如果需要的话，这些箭头前面会用方法名标注。同步方法用实心完整箭头表示，异步方法用开箭头表示。返回用虚线和开箭头表示。



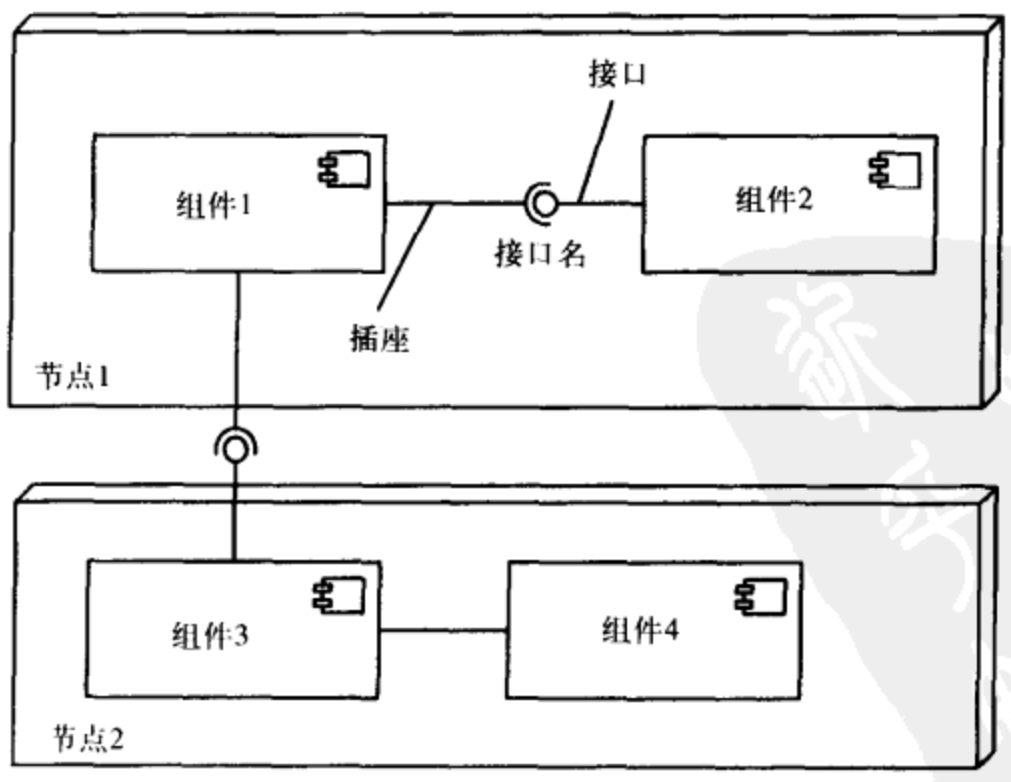
对象行为（Object Activity）。为了表示执行特定函数、过程或者方法的对象的行为，可以在连接到对象的竖条上放置方框。对象还可以执行自身的方法调用以激活其他方法。这一情形用嵌套方框（有向右的偏移）来表示。

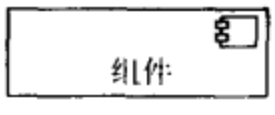


对象生命周期（Object Lifecycle）。在多数情况下，我们都假定所有相关的对象都已经存在，所以相应的方框都是画在顺序图的顶端。如果顺序图展示了对象创建，那么这是通过指向放在顺序图内的方框的箭头来表示的。如果对象不再存在，那么通过终止竖条的叉来表示。

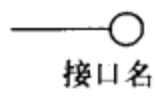
UML部署图

UML部署图描述了工件（比如组件）之间的关系。





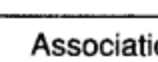
组件（Component）。组件是可部署的软件单元。它和对象不同，区分在于右上角的小构造型。



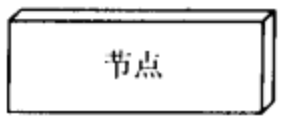
接口（Interface）。接口描述了使用组件的语法和语义。用连接到组件的线和空心圆圈来表示接口。同圆圈关联的标记表示接口名。



插座（Socket）。插座表示一个组件对另一个组件的使用。插座显示为“拥抱”接口的半圆。



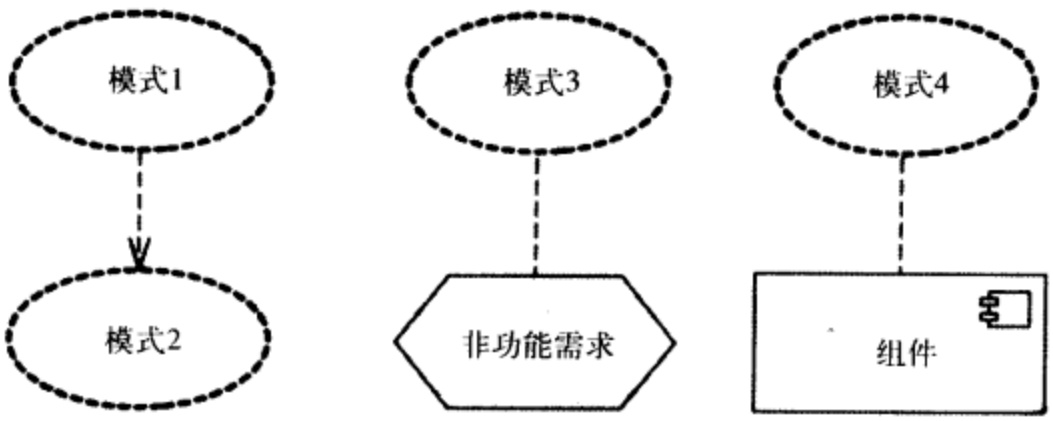
关联（Association）。关联显示为连接组件的线。虽然最好只通过精确定义的接口进行交互，但是有些情况下接口不是我们主要关心的内容，那么就可以用简单的关联来代替。



节点（Node）。节点表示运行时环境或者硬件，组件在它上面运行。它用三维框图来表示。

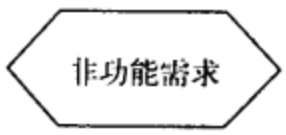
模式地图

UML建议用椭圆来在图表中表示模式。我们用模式地图来展示模式之间的关系，以及模式所涉及的其他工件（比如组件或者类）之间的关系。我们扩展了这一符号表示法，以便展示模式所解决的非功能需求。

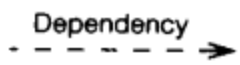




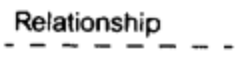
模式（Pattern）。用虚线画的椭圆表示模式。



非功能需求（Non-functional Requirement）。非功能需求用带阴影的六边形来表示。



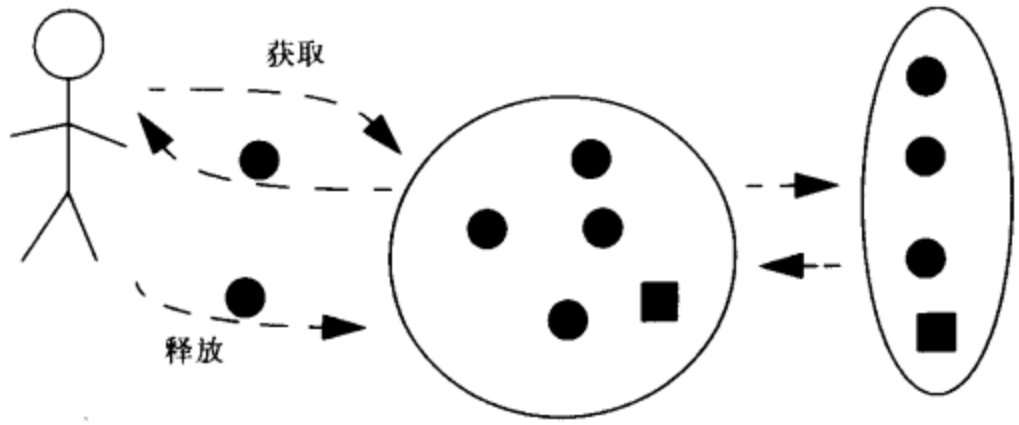
依赖（Dependency）。在模式之间绘出带箭头的虚线。在上面的图中，模式1使用了模式2来分别解决它的作用力以及解决问题。模式2的使用是可选的，因为它对于模式1不是必需的。

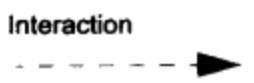


关系（Relationship）。连接到非功能需求的不带箭头的虚线表示模式解决了非功能性需求。连接到组件的不带箭头的虚线表示实现该模式需要这个组件。

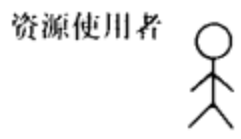
交互草图

在一些模式中，我们用下面的非正式符号表示法来描述资源使用者和模式的其他参与者之间的交互。





交互（Interaction）。交互用虚线表示，展示多个实体之间的交互。



资源使用者（Resource User）。资源使用者用一个“火柴棒小人”表示，虽然在计算机系统中资源使用者未必是人。



资源（Resource）。不同类型的资源用实心的圆或者方块表示。



容器（Container）。容器持有一个或多个资源。容器可以是资源环境，或者模式中任何持有一个或多个资源的其他参与者。

PDG

参考文献

[ACGH+96] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus: *Fault-Tolerant Telecommunication System Patterns*, in [PLoPD2], 1996

[ACM01] D. Alur, J. Crupi and D. Malks: *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2001

[Adob04] Adobe Systems Incorporated, Framemaker, `http://www.adobe.com/products/framemaker`, 2004

[Alex79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, 1979

[ALR01] A. Avizienis, J.-C. Laprie, and B. Randell: *Fundamental Concepts of Dependability*, Research Report N01145, LAAS-CNRS, April 2001

[Apac02] Apache Group, Tomcat, `http://jakarta.apache.org`, 2002

[ATT04] AT&T Corporation, On-line Directory, `http://www.anywho.com`, 2004

[Auer96] K. Auer: *Lazy Optimization: Patterns for Efficient Smalltalk Programming*, in [PLoPD2], 1996

[Aust02] D. Austerberry: *Technology of Video and Audio Streaming*, Focal Press, 2002

[BBL02] M. Baker, R. Buyya, and D. Laforenza: *Grids and Grid technologies for wide-area distributed computing*, International Journal of Software: Practice and Experience (SPE), Volume 32, Issue 15, Wiley Press, USA, `http://www.cs.mu.oz.au/~raj/papers/gridtech.pdf`, 2002

[BEA02] BEA, Weblogic Server, `http://www.bea.com/products/weblogic/server`, 2002

[BeAp02] S.P. Berczuk and B. Appelton: *Software Configuration Management Patterns*, Addison-Wesley, 2002

[Beck97] K. Beck: *Smalltalk Best Practices Patterns*, Prentice Hall, 1996

[BeCu89] K. Beck and W. Cunningham: *A Laboratory For Teaching Object-Oriented Thinking*, Proceedings of OOPSLA '89, N. Meyrowitz (ed.), special issue of SIGPLAN Notices, Volume 24, Number 10, pp. 1-6, October 1989

[BiWa04a] P. Bishop and N. Warren: *Recycle broken objects in resource pools*, Java World, `http://www.javaworld.com`, 2004

[BiWa04b] P. Bishop and N. Warren: *Lazy Instantiation*, Java World,

http://www.javaworld.com, 2004

[Bloc01] J. Bloch: *Effective Java Programming Language Guide*, Addison-Wesley, 2001

[Boeh04] H.Boehm: *A garbage collector for C and C++*, Hewlett-Packard, http://www.hpl.hp.com/personal/Hans_Boehm/gc/, 2004

[Boos04] Boost Library, Memory Pool, http://www.boost.org/libs/pool, 2004

[Borc01] J. Borchers: *A Pattern Approach to Interaction Design*, John Wiley & Sons, Inc., 2001

[Borl04] Borland, http://www.borland.com, 2004

[Busc03] F. Buschmann: *Tutorial: Patterns@Work*, OOPSLA 2003, Anaheim, USA, 2003

[CoHa01] P. Costanza and A. Haase: *The Comparand Pattern*, Proceedings of the 6th European Conference on Pattern Languages of Programs [EPLoP01], July 2001

[Cohe00] B. Cohen: *Locate & Track Pattern, Workshop: The Jini Pattern Language*, http://www.posa3.org/workshops/AdHocNetworking, OOPSLA 2000, Minneapolis, USA, 2000

[Cond04] Condor High Throughput Computing Project, http://www.cs.wisc.edu/condor, 2004

[Cope95] J.O. Coplien: *A Development Process Generative Pattern Language*, in [PLoPD1], 1995

[Cros02] J. K. Cross: *Proactive and Reactive Resource Reallocation, Workshop: Patterns in Distributed and Embedded Real-time Systems*, http://www.posa3.org/workshops/RealTimePatterns, OOPSLA 2002, Seattle, USA, 2002

[CzEi00] K. Czarnecki and U. Eisenecker: *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000

[DDN03] S. Demeyer, S. Ducasse, and O. Nierstrasz: *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2003

[DHCP04] DHCP.org, *Resources for DHCP*, http://www.dhcp.org, 2004

[Doug02] B. P. Douglass: *Real-Time Design Patterns*, Addison-Wesley, 2002

[DyLo04] P. Dyson and A. Longshaw: *Architecting Enterprise Solutions: Patterns for High-Capability Internet-based Systems*, John Wiley & Sons, Inc., 2004

[EPLoP00] M. Devos and A. Rüping (eds.): *Proceedings of the 5th European Conference on Pattern Languages of Programs*, 2000, Universitätsverlag Konstanz, 2001

[EPLoP01] A. Rüping, J. Eckstein, and C. Schwanniger (eds.): *Proceedings of the 6th*

European Conference on Pattern Languages of Programs, 2001, Universitätsverlag Konstanz, 2002

[EPLoP02] A. O'Callaghan, J. Eckstein, and C. Schwanniger (eds.): *Proceedings of the 7th European Conference on Pattern Languages of Programs*, 2002, Universitätsverlag Konstanz, 2003

[EPLoP03] K. Henney and D. Schütz (eds.): *Proceedings of the 8th European Conference on Pattern Languages of Programs*, 2003, Universitätsverlag Konstanz, 2004

[Evans04] E. Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2004

[EvCa01] M. Evans and R. A. Caras: *Hamster*, Dorling Kindersley Publishing, 2001

[Ewal01] T. Ewald: *Transactional COM+: Building Scalable Applications*, Addison-Wesley, 2001

[FeTi97] P. Feiler and W. Tichy: *Lazy Propagator*, in *Propagator: A Family of Patterns*, Proceedings of TOOLS-23, 1997

[FIK04] M. Foster, J. Ilgen, N. Kirkwood: *Tivoli Software Installation Service: Planning, Installing, and Using*, IBM RedBooks, http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246028.html?Open, 2004

[Fowl02] M. Fowler: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002

[Fowl03] M. Fowler: *UML Distilled*, 3rd edition, Addison-Wesley, 2003

[GKVI+03] G. Chen, M. Kandemir, N. Vijavkrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko: *Heap Compression for Memory-Constrained Java Environments*, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), Anaheim, USA, 2003

[GMX04] GMX, Mail, Message, More, http://www.gmx.net, 2004

[GoF95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[GaRe93] J. Gray and A. Reuter: *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993

[Grah03] I. Graham: *A Pattern Language for Web Usability*, Addison-Wesley, 2003

[Gran98] M. Grand: *Patterns in Java – Volume 1*, John Wiley & Sons, Inc., 1998

[Grid04] Grid Computing Info Centre, http://www.gridcomputing.com, 2004

[HaBr01] M. Hall and L. Brown: *Core Web Programming*, Chapter 18: *JDBC and Database Connection Pooling*, Prentice Hall, 2001

[Hanm01] R. Hanmer: *Call Processing*, Proceedings of PLoP 2001 Conference, 2001

[Hanm02] R. Hanmer: *Operations and Maintenance*, Proceedings of PLoP 2002 Conference, 2002

[HCI99] The HCI Patterns Home Page, http://www.hcipatterns.org/, established 1999

[HeBu04] K. Henney and F. Buschmann: *Beyond the Gang of Four*, Tutorial Notes of the 11th Conference on Object-Oriented-Programming, Munich Germany, 2004

[Henn01] K. Henney; *C++ Patterns: Reference Accounting*, in [EPLoP01], http://www.curbralan.com, 2001

[Henn03] K. Henney: *Factory and Disposal Methods*, VikingPLoP 2003, Bergen, Norway, http://www.curbralan.com, 2003

[Henn04] K. Henney: *One or Many*, available at http://www.curbralan.com, 2003

[HeVi99] M. Henning and S. Vinoski: *Evictor Pattern*, in *Advanced CORBA Programming with C++*, Addison-Wesley, 1999

[Hill04] Hillside Group, http://www.hillside.net, 2004

[Hope02] M. Hope: *Jaune – An ahead of time compiler for small systems*, http://jaune.sourceforge.net, 2002

[HoSm97] T. Howes and M. Smith: *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*, Macmillan Technical Publishing, 1997

[HoWo03] G. Hohpe and B. Woolf: *Enterprise Integration Patterns*, Pearson Education, 2003

[IBM02] IBM, *Ahead-Of-Time Compilation*, J9 Java Virtual Machine, http://www.ibm.com/embedded, 2002

[IBM04a] IBM, *WebSphere Application Server*, http://www.ibm.com/websphere, 2004

[IBM04b] IBM, *The Eclipse project*, http://www.eclipse.org, 2004

[Iona04] IONA, http://www.iona.com, 2004

[IrDA04] Infrared Data Association (IrDA), http://www.irda.org, 2004

[JAP02] Z. Juhasz, A. Andics, and S. Pota: *Towards a Robust and Fault-Tolerant Multicast Discovery Architecture for Global Computing Grids*, in P. Kacsuk, D. Kranzlmüller, Zs. Nemeth, J. Volkert (Eds.): *Distributed and Parallel Systems – Cluster and Grid Computing*, Proceedings of the 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems, Kluwer Academic Publishers, The Kluwer International Series in Engineering and Computer Science, Volume 706, Linz, Austria, pp. 74-81, 2002

[Jain01] P. Jain: *Evictor Pattern*, Proceedings of the 8th Conference on Pattern Languages of Programs, Allerton Park, Illinois, USA, 2001

[Jain02] P. Jain: *Coordinator Pattern*, Proceedings of the 7th European Conference on Pattern Languages of Programs [EPLoP02], 2002

[JaKi00] P. Jain and M. Kircher: *Leasing Pattern*, Proceedings of the 7th Conference on Pattern Languages of Programs, Allerton Park, Illinois, USA, 2000

[JaKi02] P. Jain and M. Kircher: *Partial Acquisition*, Proceedings of the 9th Conference on Pattern Languages of Programs, Allerton Park, Illinois, USA, 2002

[JoLi96] R. Jones, and R. Lins: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, Inc., 1997

[JXTA04] JXTA Project, `http://www.jxta.org`, 2004

[KiJa00a] M. Kircher and P. Jain: *Lookup Pattern*, Proceedings of the 5th European Conference on Pattern Languages of Programs [EPLoP00], 2000

[Kirc01] M. Kircher: *Lazy Acquisition Pattern*, Proceedings of the 6th European Conference on Pattern Languages of Programs [EPLoP01], 2001

[Kirc02] M. Kircher: *Eager Acquisition Pattern*, Proceedings of the 7th European Conference on Pattern Languages of Programs [EPLoP02], 2002

[KiJa02] M. Kircher and P. Jain: *Pooling Pattern*, Proceedings of the 7th European Conference on Pattern Languages of Programs [EPLoP02], 2002

[KiJa03a] M. Kircher and P. Jain: *Caching Pattern*, Proceedings of the 8th European Conference on Pattern Languages of Programs [EPLoP03], 2003

[KiJa03b] M. Kircher and P. Jain: *Resource Lifecycle Manager Pattern*, Proceedings of the 8th European Conference on Pattern Languages of Programs [EPLoP03], 2003

[KiJa04] M. Kircher and P. Jain: *JinACE*, `http://www.posa3.org/jinace`, 2004

[KiSa92] J. J. Kistler and M. Satyanarayanan: *Disconnected Operation in the Coda File System*, ACM Transactions on Computer Systems, Volume 10, Issue 1, 1992

[KoMa95] F. Kon and A. Mandel: *SODA: A Lease-Based Consistent Distributed File System*, Proceedings of the 13th Brazilian Symposium on Computer Networks, 1995

[Ladd03] R. Laddad: *AspectJ in Action*, Practical Aspect-Oriented Programming, Manning Publications, 2003

[Lapr85] J.C. Laprie: *Dependability: Basic Concepts and Terminology*, Proceedings of the 15th International Symposium on Fault-Tolerant Computing, 1985

[Lea99] D. Lea: *Concurrent Programming in Java: Design Principles and Pattern*,

Addison-Wesley, 1999

[LiCa87] M. A. Linton and P. R. Calder: *The Design and Implementation of InterViews*, Proceedings of the USENIX C++ Workshop, November 1987

[LGS01] D. L. Levine, C. D. Gill, and D. C. Schmidt: *Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction*, in *Design Patterns in Communications Software*, Linda Rising, ed., Cambridge University Press, 2001

[Lieb01] J. Liebenau: *Abstract Manager*, Proceedings of the 8[th] Conference on Pattern Languages of Programs, Allerton Park, Illinois, USA, 2001

[LoLa01] F. Longoni and A. Laensisalmi: *Radio Access Network Architecture*, in *WDCMA for UMTS – Radio Access for Third Generation Mobile Communications*, John Wiley & Sons, Inc., 2001

[Luce03] iMerge, *iMerge Element Manager User's Guide*, Lucent Technologies, 2003

[Lynu04] LYNUXWORKS, LynxOS, `http://www.lynuxworks.com/rtos`, 2004

[MaAs02] A. Martelli, D. Ascher: *Python Cookbook*, O'Reilly, 2002

[Macr04] Macrovision, *FLEXlm Software Licensing System*, `http://www.macrovision.com/products/flexlm`, 2004

[Mesz95] G. Meszaros: *Half-Object Plus Protocol*, in [PLoPD1], 1995

[Metz02] S.J. Metzker: *Design Patterns Java Workbook*, Addison-Wesley, 2002

[Meye98] S. Meyers: *Effective C++*, 2nd Edition, Addison-Wesley, 1997

[Micr04] Microsoft Internet Explorer, `http://www.microsoft.com/windows/ie`, 2004

[MoOh97] P. Molin and L. Ohlsson: *The Points and Deviations Pattern Language of Fire Alarm Systems – Lazy State Pattern*, in [PLoPD3], 1997

[Mozi04] Mozilla.org, Mozilla Browser, `http://www.mozilla.org/`, 2004

[MSN04] MSN Hotmail, `http://hotmail.msn.com`, 2004

[Nets04] Netscape Browser, `http://www.netscape.com`, 2004

[NewM04] NewMonics Inc., *PERC JVM*, `http://www.newmonics.com/perc/info.shtml`, 2004

[Newp04] B. Newport: *Implementing a Data Cache using Readers And Writers*, TheServerSide.COM, `http://www.theserverside.com/`, 2004

[Nock03] C. Nock: *Data Access Patterns: Database Interactions in Object-Oriented Applications*, Addison-Wesley, 2003

[NoWe00] J. Noble and C. Weir: *Variable Allocation Pattern*, in *Small Memory Software:*

Patterns for Systems with Limited Memory, Addison-Wesley, 2000

[NW088] M. N. Nelson, B. B. Welch, and J. Ousterhout: *Caching in the Sprite Network Operating System*, ACM Transactions on Computer Systems (TOCS), Volume 6, Issue 1, 1988

[OCI04] Object Computing Interactive, *The ACE ORB*, http://www.theaceorb.com, 2004

[OMG04a] Object Management Group, *Common Object Request Broker Architecture (CORBA/IIOP)*, http://www.omg.org/technology, 2004

[OMG04b] Object Management Group, *Real-Time CORBA 1.0 Specification*, Chapter 24.15, *Thread pools*, http://www.omg.org/technology, 2004

[OMG04c] Object Management Group, *CORBA Naming Service*, http://www.omg.org/technology, 2004

[OMG04d] Object Management Group, *CORBA Component Model Specification (CCM)*, http://www.omg.org/technology, 2004

[OMG04e] Object Management Group, *Object Transaction Service*, http://www.omg.org/technology, 2004

[OMG04f] Object Management Group, *CORBA Trading Object Service*, http://www.omg.org/technology, 2004

[Orac03] Oracle, *Oracle Application Server: Java Object Cache*, http://otn.oracle.com/products/ias/joc, 2003

[Peda04] Pedagogical Patterns Project, http://www.pedagogicalpatterns.org, 2004

[PeSo97] D. Petriu, G. Somadder: *A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers*, Proceedings of the 2nd European Conference on Pattern Languages of Programs, Kloster Irsee, Germany, 1997

[PLoPD1] J.O. Coplien and D.C. Schmidt (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995 (a book publishing the reviewed Proceedings of the First International Conference on Pattern Languages of Programming, Monticello, Illinois, 1994)

[PLoPD2] J.O. Coplien, N. Kerth, and J. Vlissides (eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995)

[PLoPD3] R.C. Martin, D. Riehle, and F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997 (a book publishing selected papers from the Third International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1996, the First European Conference

on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1996, and the Telecommunication Pattern Workshop at OOPSLA '96, San Jose, California, USA, 1996)

[PLoPD4] N. Harrison, B. Foote, and H. Rohnert (eds.): *Pattern Languages of Program Design 4*, Addison-Wesley, 1999 (a book publishing selected papers from the Fourth and Fifth International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1997 and 1998, and the Second and Third European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1997 and 1998)

[Pont01] M. J. Pont: *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*, Addison-Wesley, 2001

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, Inc., 1996

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann: *Pattern-Oriented Software Architecture – Patterns for Concurrent and Distributed Objects*, John Wiley & Sons, Inc., 2000

[POSA4] F. Buschmann and K. Henney: *Pattern-Oriented Software Architecture – On Patterns and Pattern Languages*, John Wiley & Sons, Inc., to be published 2005

[Pree94] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

[Pryc02] N. Pryce: *Eager Compilation and Lazy Evaluation*, http://www.doc.ic.ac.uk/~np2/patterns, 2002

[Ramm02] I. Rammer: *Advanced .NET Remoting*, APress, 2002

[Rees92] T. Reenskaug: *Intermediate Smalltalk, Practical Design and Implementation*, Tutorial, TOOLS Europe '92, Dortmund, 1992

[Rich02] J. M. Richter: *Applied Microsoft .NET Framework Programming*, Microsoft Press, 2002

[Risi00] L. Rising: *The Pattern Almanac 2000*, Addison-Wesley, 2000

[RoDe90] J. T. Robinson, M. V. Devarakonda: *Data Cache Management Using Frequency-Based Replacement*, In Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, 1990

[SaCa96] A. Sane and R. Campbell: *Resource Exchanger*, in [PLoPD2], 1996

[Sari02] T. Saridakis: *A System of Patterns for Fault Tolerance*, Proceedings of the 7th European Conference on Pattern Languages of Programs [EPLoP02], 2002

[Sari03] T. Saridakis: *Design Patterns for Fault Containment*, Proceedings of the 8th

European Conference on Pattern Languages of Programs [EPLoP03], 2003

[Schm98] D. C. Schmidt, D. L. Levine, and S. Mungee: *The Design and Performance of Real-Time Object Request Brokers*, Computer Communications, Volume 21, Pages 294-324, Elsevier, 1998

[Schm02] D. C. Schmidt and S. Huston: *C++ Network Programming: Mastering Complexity with ACE and Patterns*, Addison-Wesley, 2002

[Schm03a] D. C. Schmidt and S. Huston: *C++ Network Programming: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, 2003

[Schm04] D. C. Schmidt: *Smart Proxies*, http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/Smart_Proxies.html, 2004

[Siem03] Siemens AG, *UMTS Solutions*, http://www.siemens.ie/mobile/UMTS, 2004

[SFHB04] M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann: *Security Patterns*, John Wiley & Sons, Inc., to be published in 2004

[ShaTr01] A. Shalloway and J.R. Trott: *Design Patterns Explained*, Addison-Wesley, 2001

[Shaw01] A. C. Shaw: *Real-Time Systems and Software*, John Wiley & Sons, Inc., 2001

[Shif04] ShiftOne Object Cache, http://sourceforge.net/projects/jocache, 2004

[SMFG01] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A.Gokhale: *Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers*, Journal of Real-time Systems, Kluwer, Volume 21, Number 2, 2001.

[Smit82] A. J. Smith: *Cache Memories*, Journal of ACM Computing Surveys, Volume 14 Issue 3, 1982

[Smit85] A. J. Smith: *Disk Cache – Miss Ratio Analysis and Design Considerations*, ACM Transactions on Computer Systems (TOCS), Volume 3, Issue 3, 1985

[Somm98] P. Sommerlad: *Manager Pattern*, in [PLoPD3], 1998

[Somm02] P. Sommerlad: *Performance Patterns*, Proceedings of the 7th European Conference on Pattern Languages of Programs [EPLoP02], 2002

[Squi04] Squid Web Proxy Cache, http://www.squid-cache.org, 2004

[Stal00] M. Stal: *Activator Pattern*, http://www.stal.de/articles.html, 2000

[Stal00] M. Stal: *Deployer Pattern*, http://www.stal.de/Downloads/docpatterns.pdf, 2000

[Stev03] W. R. Stevens: *Unix Network Programming, Volume 1: The Sockets Networking API*, Third Edition, Prentice Hall, 2003

[Sun04a] Sun Microsystems, *Java*, http://java.sun.com/, 2004

[Sun04b] Sun Microsystems, *Java2 Enterprise Edition (J2EE)*, http://java.sun.com/j2ee/, 2004

[Sun04c] Sun Microsystems, *Jini*, http://www.sun.com/jini, 2004

[Sun04d] Sun Microsystems, *Java Transaction Service*, http://java.sun.com/products/jts, 2004

[Sun04e] Sun Microsystems, *Java Authentication and Authorization Service*, http://java.sun.com/products/jaas/, 2004

[Sun04f] Sun Microsystems, *Java Naming and Directory Interface*, http://java.sun.com/products/jndi/, 2004

[Sun04g] Sun Microsystems, *Java Remote Method Invocations (RMI)*, http://java.sun.com/products/jdk/rmi/, 2004

[Sun04h] *Solaris 9 Linker and Libraries Guide*, Section on *Lazy Loading of Dynamic Dependencies*, http://docs.sun.com, 2004

[Sun04i] Sun Microsystems, *Java Connector Architecture (JCA)*, http://java.sun.com/j2ee/connector, 2004

[Sun04j] Sun Microsystems, *Mobile Information Device Profile for Java 2 Micro Edition, Version 2.0 – Over The Air User Initiated Provisioning*, http://java.sun.com/products/midp, 2004

[Symb03] Symbian Ltd, *Symbian OS – the mobile operating system*, http://www.symbian.com, 2004

[Tane01] A. S. Tanenbaum: *Modern Operating Systems*, 2nd Edition, Prentice Hall, 2001

[Tane02] A. S. Tanenbaum: *Computer Networks*, 4th Edition, Prentice Hall, 2002

[Thom99] S. Thompson: *Haskell: The Craft of Functional Programming*, 2nd Edition, Addison-Wesley, 1999

[TMQH+03] D. Trowbridge, D. Mancini, D. Quick, G. Hohpe, J. Newkirk, and D. Lavigne: Microsoft, *Enterprise Solution Patterns Using Microsoft .NET*, http://www.microsoft.com/resources/practices, 2003

[ToHu00] D. Thomas and A. Hunt: *Programming Ruby: A Pragmatic Programmers Guide*, Addison-Wesley, 2000

[TwAl83] J. Twing and G. Alpharetta: *The Receptionist*, Macmillan McGraw Hill, 1983

[UDDI04] Universal Description, Discovery, and Integration of Web Services (UDDI), http://uddi.org, 2004

[UPnP04] Universal Plug 'n Play (UPnP) Forum, http://www.upnp.org, 2004

[VBW97] T. E. Vollmann, W. L. Berry, D. C. Whybark: *Manufacturing Planning and Control Systems*, McGraw-Hill Trade, 4th Edition, 1997

[VKZ04] M. Völter, M. Kircher, and U. Zdun: *Remoting Patterns – Patterns for Enterprise, Realtime and Internet Middleware*, John Wiley & Sons, Inc., to be published in 2004

[VPLoP02] P. Hruby and K.E. Sørensen (eds.): *Proceedings of the First Nordic Conference on Pattern Languages of Programs 2002*, Microsoft Business Solutions, 2003

[VSW02] M. Völter, A. Schmid, E. Wolff: *Server Component Patterns – Component Infrastructures Illustrated with EJB*, John Wiley & Sons, Inc., 2002

[Wall97] E. Wallingford: *Sponsor-Selector*, in [PLoPD3], 1997

[Wind04] Windriver, VxWorks, http://www.windriver.com, 2004

[Wisc00] M. Wischy: *Pick and Verify Pattern*, Workshop: *The Jini Pattern Language*, http://www.posa3.org/workshops/AdHocNetworking, OOPSLA 2000, Minneapolis, USA, 2000

[W3C04] World Wide Web Consortium (W3C), http://www.w3.org, 2004

[YaAm03] S. Yacoub and H.H. Ammar: *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*, Addison-Wesley, 2003

[YBS99] H. Yu, L. Breslau, and S. Shenker: *A Scalable Web Cache Consistency Architecture*, Computer Communication Review, ACM SIGCOMM, Volume 29, Number 4, October 1999

[Zero04a] ZeroC Inc., *Internet Communication Engine (Ice)*, http://www.zeroc.com, 2004

[Zero04b] Zero G, *InstallAnywhere*, http://www.zerog.com, 2004

[3GPP04] *3rd Generation Partnership Project* (3GPP), http://www.3gpp.org, 2004

[General Information]
书名=设计系列　面向模式的软件体系结构　第3卷
作者=MICHAEL KIRCHER　PRASHANT JAIN著　鲍志云译
丛书名=软件工程技术丛书
页数=169
SS号=11474873
出版日期=2005年09月第1版
出版社=机械工业出版社
尺寸=24cm
原书定价=29
主题词=软件(学科：系统结构)软件　系统结构
参考文献格式=Michael Kircher, Prashant Jain著;鲍志云译.面向模式的软件体系结构 卷3 patterns for resource management Volume 3.北京市:机械工业出版社,2005.09.
内容提要=本书用模式来展现在系统中实现有效的资源管理所需的技术，提供了对资源管理主题的介绍，讲解了资源获取、资源生命周期、资源释放中涉及的相关模式，这同典型的资源生命周期相对应。

封面
书名
版权
前言
目录译者序Ｆｒａｎｋ　Ｂｕｓｃｈｍａｎｎ序Ｓｔｅｖｅ　Ｖｉｎｏｓｋｉ序前言作者简介第１章　导引
１．１　资源管理概览
１．２　资源管理的范围
１．３　模式的使用
１．４　资源管理中的模式
１．５　相关工作
１．６　模式格式
第２章　资源获取
２．１　Ｌｏｏｋｕｐ模式
２．２　Ｌａｚｙ　Ａｃｑｕｉｓｉｔｉｏｎ模式
２．３　Ｅａｇｅｒ　Ａｃｑｕｉｓｉｔｉｏｎ模式
２．４　Ｐａｒｔｉａｌ　Ａｃｑｕｉｓｉｔｉｏｎ模式
第３章　资源生命周期
３．１　Ｃａｃｈｉｎｇ模式
３．２　Ｐｏｏｌｉｎｇ模式
３．３　Ｃｏｏｒｄｉｎａｔｏｒ模式
３．４　Ｒｅｓｏｕｒｃｅ　Ｌｉｆｅｃｙｃｌｅ　Ｍａｎａｇｅｒ模式
第４章　资源释放
４．１　Ｌｅａｓｉｎｇ模式
４．２　Ｅｖｉｃｔｏｒ模式
第５章　资源管理准则
第６章　案例分析——自组网络计算
６．１　概览
６．２　动机
６．３　解决方案
第７章　案例分析——移动网络
７．１　概览
７．２　动机
７．３　解决方案
第８章　模式的过去、现在和未来
８．１　过去的四年
８．２　模式的现状如何
８．３　模式明天将走向何方
８．４　关于模式未来的简短声明
第９章　结语
引用到的模式
符号表示法
参考文献
致谢